

Л.П.БАБЕНКО, К.М.ЛАВРИЩЕВА

# ОСНОВИ **ПРОГРАМНОЇ ІНЖЕНЕРІЇ**

НАВЧАЛЬНИЙ ПОСІБНИК



Знання

---

Серія "ВИЩА ОСВІТА ХХІ СТОЛІТТЯ"

ІНСТИТУТ ПРОГРАМНИХ СИСТЕМ НАН УКРАЇНИ  
КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ  
імені ТАРАСА ШЕВЧЕНКА

Л.П.БАБЕНКО, К.М.ЛАВРІЩЕВА

# ОСНОВИ ПРОГРАМНОЇ ІНЖЕНЕРІЇ

НАВЧАЛЬНИЙ ПОСІБНИК

*Рекомендовано  
Міністерством освіти і науки України  
для студентів вищих навчальних закладів*

НБ ПНУС



637782



Київ  
"Знання"  
2001

---

УДК 681.3.06 (075.8)  
ББК 32.973я73  
Б12

Рекомендовано Міністерством освіти і науки України (лист  
№ 14/18.2—247 від 13 березня 2001 р.)

Затверджено до друку Вченою радою факультету кібернетики  
Київського національного університету імені Тараса Шевченка  
(протокол № 9 від 29 травня 2000 р.)

**Рецензенти:**

О.А. Павлов, доктор технічних наук, професор, декан факультету  
інформатики та обчислювальної техніки Національного технічного уні-  
верситету України (КПІ), Заслужений діяч науки і техніки України;

В.Н. Редько, доктор фізико-математичних наук, професор, академік  
НАН України, завідувач кафедри теоретичного програмування Київ-  
ського національного університету імені Тараса Шевченка;

А.А. Летичевський, доктор фізико-математичних наук, професор,  
член-кореспондент НАН України, завідувач відділу Інституту кібер-  
нетики НАН України

**Бабенко Л.П., Лавріщева К.М.**

Б12 Основи програмної інженерії: Навч. посіб. — К.: Т-во “Знан-  
ня”, КОО, 2001.—269 с. — (Вища освіта XXI століття).

ISBN 966-620-095-3

Маючи багаторічний досвід створення складних програмних си-  
стем та читання лекцій з програмної інженерії у Київському націо-  
нальному університеті імені Тараса Шевченка, автори провели си-  
стематизацію великого обсягу сучасних наукових джерел відповідно  
до напрацювань SWEBOK та визначили основи програмної інженерії,  
якими має володіти фахівець. Ця книга є загальним введенням до су-  
часних інженерних об'єктно-орієнтованих методів розробки програм-  
них систем. У ній розглянуто широке коло питань, які виникають на шля-  
ху від усвідомлення замовником своїх потреб до створення праце-  
здатного програмного забезпечення.

Автори адресують книгу менеджерам колективів програмістів,  
які бажають підвищити рівень кваліфікації своєї організації, студен-  
там та аспірантам вищих навчальних закладів, розробникам навчаль-  
них курсів для закладів, які готують спеціалістів з інформатики, фа-  
хівцям різних проблемних галузей, які хочуть бути грамотними за-  
мовниками сучасних комп'ютерних систем.

УДК 681.3.06 (075.8)  
ББК 32.973я73

© Л.П. Бабенко, К.М. Лавріщева, 2001  
© Київська обласна організація това-  
риства “Знання” України, 2001

## ЗМІСТ

### ПЕРЕДМОВА ..... 9

### Глава 1. ПРОГРАМНА ІНЖЕНЕРІЯ ЯК ФАХ ..... 13

Контрольні запитання і завдання ..... 16

### Глава 2. БАЗОВІ ПОНЯТТЯ ПРОГРАМНОЇ ІНЖЕНЕРІЇ ..... 17

Контрольні запитання і завдання ..... 21

### Глава 3. ІНЖЕНЕРІЯ ВИМОГ ..... 22

- 3.1. Інженерія вимог як процес ..... 22
  - 3.2. Концептуальне моделювання проблеми ..... 26
    - 3.2.1. Мета ..... 26
    - 3.2.2. Онтологія домену ..... 26
    - 3.2.3. Моделі динамічних явищ домену ..... 28
    - 3.2.4. Модель алгоритмів ..... 29
  - 3.3. Об'єктно-орієнтована інженерія вимог ..... 29
  - 3.4. Метод інженерії вимог С. Шлеєр та С. Меллора ..... 34
    - 3.4.1. Інформаційна модель або онтологія домену ..... 34
    - 3.4.2. Модель станів ..... 40
    - 3.4.3. Модель процесів ..... 44
    - 3.4.4. Продукти інженерії вимог за методом  
С. Шлеєр та С. Меллора ..... 47
  - 3.5. Метод інженерії вимог І. Джекобсона ..... 47
    - 3.5.1. Концепція моделі сценаріїв для збирання вимог ..... 47
    - 3.5.2. Модель аналізу вимог. Визначення об'єктів ..... 56
    - 3.5.3. Продукти інженерії вимог за методом І. Джекобсона ... 60
- Контрольні запитання і завдання ..... 60

### Глава 4. ПРОЕКТУВАННЯ ПРОГРАМНИХ СИСТЕМ ..... 62

- 4.1. Проектування як процес ..... 62
- 4.2. Концептуальне проектування ..... 63
  - 4.2.1. Уточнення даних ..... 63
  - 4.2.2. Уточнення інтерфейсів ..... 65
  - 4.2.3. Уточнення функцій обробки даних ..... 66
  - 4.2.4. Уточнення нефункціональних вимог ..... 67

4.3.	Архітектурне проектування .....	68
4.4.	Технічне проектування .....	72
	Контрольні запитання і завдання .....	76

<b>Глава 5.</b>	<b>МЕТОД UML ЯК ПОТЕНЦІЙНИЙ СТАНДАРТ ЗАСОБІВ МОДЕЛЮВАННЯ В ПРОГРАМНІЙ ІНЖЕНЕРІЇ .....</b>	<b>77</b>
5.1.	Концепція методу .....	77
5.2.	Діаграми класів .....	79
5.3.	Діаграми сценаріїв .....	83
5.4.	Діаграми моделювання поведінки системи .....	83
5.5.	Діаграми послідовності .....	84
5.6.	Діаграми співробітництва .....	85
5.7.	Діаграми діяльності .....	86
5.8.	Діаграми станів .....	86
5.9.	Діаграми реалізації .....	88
5.9.1.	Діаграми компонент .....	88
5.9.2.	Діаграми розміщення .....	88
5.10.	Пакети в UML .....	88
	Контрольні запитання і завдання .....	91

<b>Глава 6.</b>	<b>ТРАНСФОРМАЦІЯ ПРОЕКТУ В ПРОГРАМУ .....</b>	<b>92</b>
	Контрольні запитання і завдання .....	93

<b>Глава 7.</b>	<b>ТЕСТУВАННЯ ПРОГРАМ ТА СИСТЕМ .....</b>	<b>94</b>
7.1.	Методи тестування програм .....	95
7.1.1.	Статичні методи тестування .....	96
7.1.2.	Динамічні методи тестування .....	101
7.1.3.	Функціональне тестування .....	104
7.2.	Методи доведення правильності програм .....	106
7.2.1.	Техніка формального доведення .....	108
7.2.2.	Техніка символічного виконання .....	113
7.2.3.	Верифікація та валідація .....	114
7.3.	Об'єкти тестування .....	117
7.3.1.	Тестування окремого елемента .....	119
7.3.2.	Перевірка коду .....	120
7.3.3.	Інспекції коду .....	120
7.4.	Класифікація відмов та помилок .....	121
7.4.1.	Помилки та причини появи їх на етапах ЖЦ .....	122
7.4.2.	Зв'язок помилки з відмовою .....	124
7.4.3.	Джерела помилок .....	125
7.5.	Тести програм і систем .....	129
7.5.1.	Види тестів .....	129
7.5.2.	Команда тестувачів .....	132

7.5.3.	План тестування .....	133
7.6.	Засоби автоматизації процесу тестування .....	134
	Контрольні запитання і завдання .....	136

<b>Глава 8.</b>	<b>СУПРОВОДЖЕННЯ ПРОГРАМНИХ СИСТЕМ .....</b>	<b>138</b>
	Контрольні запитання і завдання .....	141

<b>Глава 9.</b>	<b>АНАЛІЗ І ДОСЯГНЕННЯ ЯКОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ .....</b>	<b>142</b>
9.1.	Визначення якості програмного забезпечення .....	143
9.2.	Організація робіт для досягнення якості програмного забезпечення .....	144
9.3.	Модель якості програмного забезпечення .....	145
9.3.1.	Функціональність .....	147
9.3.2.	Надійність .....	149
9.3.3.	Зручність застосування .....	151
9.3.4.	Ефективність .....	152
9.3.5.	Супроводжуваність .....	152
9.3.6.	Переносність .....	153
9.4.	Методи визначення значень показників якості .....	153
9.5.	Метрики якості програмного забезпечення .....	155
9.5.1.	Метрики програмного продукту .....	156
9.5.2.	Метрики процесів .....	157
9.5.3.	Вимірювання метрик якості .....	158
9.6.	Аналіз якості програмного забезпечення .....	160
9.7.	Стандарти підтримки якості програмного забезпечення .....	162
9.7.1.	Конкретизація моделі якості .....	163
9.7.2.	Керування якістю .....	164
9.8.	Оцінка надійності програмного забезпечення .....	168
9.8.1.	Визначення мети і завдань забезпечення надійності на етапах життєвого циклу .....	168
9.8.2.	Моделі та методи оцінки надійності .....	171
9.9.	Експериментальні підходи до оцінки надійності .....	177
9.10.	Методи оцінки рівня якості програмного забезпечення ...	178
	Контрольні запитання і завдання .....	181

<b>Глава 10.</b>	<b>КЕРУВАННЯ КОНФІГУРАЦІЯМИ ПРОГРАМНИХ СИСТЕМ .....</b>	<b>183</b>
10.1.	Зміст процесу керування конфігурацією .....	183
10.2.	Керування конфігурацією як процес .....	185
10.3.	Ідентифікація конфігурації .....	189
10.4.	Організація процесу змін .....	190
10.5.	Конфігураційний облік .....	193

10.6.	Аудит конфігурації .....	194
10.7.	Керування версіями та передача версій замовникам ...	194
10.8.	Управління документацією та супроводженням .....	196
	Контрольні запитання і завдання .....	198

## Глава 11. МЕНЕДЖМЕНТ ІНЖЕНЕРІЇ ПРОГРАМНИХ СИСТЕМ

11.1.	Аналіз стану .....	199
11.2.	Планування .....	202
11.3.	Персональна відповідальність у проекті .....	205
11.4.	Проведення оцінок проекту .....	208
11.5.	Керування ризиком .....	210
11.6.	План проекту .....	213
11.7.	Моделі процесу ведення проекту .....	214
11.8.	Метрики менеджменту .....	217
	Контрольні запитання і завдання .....	218

## Глава 12. ПОВТОРНЕ ВИКОРИСТАННЯ В ПРОГРАМНІЙ ІНЖЕНЕРІЇ

12.1.	Зміст проблеми .....	220
12.2.	Визначальні властивості ПВК та їх типові сполучення .....	222
12.3.	Поширені в користуванні категорії ПВК .....	225
12.3.1.	Мови програмування високого рівня .....	225
12.3.2.	Компоненти вихідного коду .....	226
12.3.3.	Класи об'єктів та абстрактні класи .....	227
12.3.4.	Абстрактні архітектури програмних систем .....	228
12.3.5.	Генератори прикладних застосувань .....	229
12.3.6.	Абстрактні домени .....	231
12.3.7.	Патерни .....	232
12.3.8.	Каркаси (Frameworks) .....	235
12.4.	Створення повторно використовуваних компонент .....	236
12.4.1.	Вияв потенційних ПВК .....	236
12.4.2.	Специфікація варіантності вимог .....	239
12.4.3.	Конкретизація варіантності вимог .....	242
12.4.4.	Проектування ПВК .....	246
12.5.	Інформаційне забезпечення повторного використання ...	249
	Контрольні запитання і завдання .....	252

## Глава 13. СТАНДАРТИ ПРОГРАМНОЇ ІНЖЕНЕРІЇ

	Контрольні запитання і завдання .....	254
ДОДАТОК А.	Етичний кодекс програмної інженерії .....	259
ДОДАТОК Б.	Глосарій .....	261

## ПЕРЕДМОВА

Термін *програмна інженерія* (Software Engineering) вперше вжито в 1968 році на науковій конференції НАТО, чим було засвідчено, що створення програмного забезпечення досягло такого ступеня розвитку, коли можна застосовувати інженерні технології. З того часу програмне забезпечення проникло в усі сфери людського буття, а його розробка стала справді масовою діяльністю.

Інструменти підтримки розробки програмних систем зробили величезний стрибок у своєму розвитку, і тепер звичайною практикою стали: створення програмних систем як композиції готових компонент, побудова інтерфейсів користувача шляхом маніпуляцій з «мишею», візуалізація процесів та продуктів розробки.

Водночас, залишаються актуальними питання організації розробки великих систем, шляхів та засобів досягнення їхньої високої якості й відповідності до потреб замовника. Саме володіння методами розв'язання таких проблем дає надію на вдалу реалізацію великих програмних систем та їхню конкурентоздатність.

Отже, важливо об'єднати досвід попередніх розробок як певну суму професійних знань, які могла б опановувати переважна більшість виконавців розробок, і тоді передовий досвід став би надбанням широкого кола професіоналів.

Саме таких зусиль тепер докладають професійні об'єднання та провідні виробники програмних систем для визначення ядра професійних знань, що становлять предмет програмної інженерії як фаху. Це ядро вони назвали *Software Engineering Body of Knowledge (SWEBOK)*. Першим результатом їхньої діяльності є визначення головних розділів ядра знань та відповідних тематичних рубрик.

Нині проблеми і питання програмної інженерії висвітлюються в низці статей, в окремих параграфах і главах закордонних монографій. Проте монографії, яка охоплювала б усі основні питання цієї проблематики, в українській і російськомовній літературі немає. Це й спонукало нас до спроби заповнити цю прогалину і написати посібник.

Задумуючи цей посібник, ми керувалися таким принципом: розглянути основи програмної інженерії для того, щоб читачі мали змогу познайомитися з досягненнями в галузі програмної інженерії,

оскільки першоджерела практично малодоступні.

До посібника включено сучасні методи інженерії, що пройшли практичну апробацію під час розробки великих програмних систем. Результати цієї діяльності автори опублікували у низці статей і монографій, а також як матеріали в електронному варіанті для читання лекцій у Київському національному університеті ім. Тараса Шевченка.

Водночас посібник є спробою суцільного викладу основних положень програмної інженерії. Його сплановано так, що кожний розділ відповідає послідовності процесів розробки сучасних програмних систем, починаючи від постановки завдання і закінчуючи виготовленням програмної системи з перевіркою її на тестах й оцінкою показників якості, які задовольняють замовника.

Автори посібника виклали зміст програмної інженерії відповідно до напрацювань зі створення SWEBOOK. Добираючи матеріал, ми керувалися прагненням висвітлити концепції та методи, перевірені практикою, обминаючи націлені на майбутнє теоретичні дослідження, які ще не підтвердили своєї придатності для широкого впровадження.

Ця книга є загальним введенням у сучасні інженерні методи розробки програмних систем. Оскільки немає простих розв'язань для складних питань, тому, не шукаючи панацеї, ми зробили спробу розглянути широке коло питань, які виникають і потребують відповідей на шляху від розуміння замовником його потреб до створення працездатного програмного забезпечення, що задовольняє згадані вище потреби.

Книга має 13 глав. До кожної глави додаються контрольні запитання і завдання. Відповідаючи на запропоновані запитання та розв'язуючи завдання, читач зможе ще раз зосередити свою увагу на центральних аспектах кожної глави і проконтролювати своє розуміння викладених знань. Нижче подається короткий зміст глав.

#### Глава 1. Програмна інженерія як фах.

Подається визначення програмної інженерії, її специфіка як інженерної діяльності зі створення програмних систем. Обговорюються питання становлення фаху, зміст його головних напрямів дій, зв'язок з іншими комп'ютерними дисциплінами.

#### Глава 2. Базові поняття програмної інженерії.

Наводяться головні елементи еталонної моделі програмної інженерії та заснованих на них моделей життєвого циклу розробки програмних систем.

#### Глава 3. Інженерія вимог.

Обґрунтовується інженерія вимог як визначальна стадія життєвого циклу, мета якої — сформулювати договір між замовником і розробником мовою, ступінь формалізації котрої не заважає досягненню домовленостей. Аналізуються й зіставляються кілька конструктивних підходів, перевірених широким використанням, які претендують на роль стандартів у майбутньому, але майже невідомі вітчизняним

розробникам через брак доступних публікацій. Розглянуто об'єктно-орієнтовані методи аналізу вимог та аналізу проблемних областей.

#### Глава 4. Проектування програмних систем.

Розглядаються проблеми трансформації вимог замовника до розробки в проектні рішення з визначення структури та особливостей функціонування майбутньої системи.

Глава 5. Метод UML як потенційний стандарт засобів моделювання в програмній інженерії.

Розглядаються концепції та елементи сучасного методу, який набув широкого визнання та застосування. Мова методу призначена для специфікації, візуалізації, конструювання та документування артефактів програмних систем.

#### Глава 6. Трансформація проекту в програму.

Подається загальна схема деталізації проекту під час створення вихідної програми.

#### Глава 7. Тестування програм та систем.

Наводяться методи тестування, верифікації та валідації програм, наводиться класифікація помилок та відмов у програмах, розглядаються різновиди тестів та елементів тестування, визначається роль команди тестувачів.

#### Глава 8. Супроводження програмних систем.

Розглядаються питання забезпечення еволюції програмних систем як головного фактора їхньої життєдіяльності та життєздатності.

Глава 9. Аналіз і досягнення заданої якості програмного забезпечення.

Подано визначення поняття якості програмного забезпечення, досліджено моделі та методи подання вимог до якості, стратегії та оцінок досягнення показників якості впродовж життєвого циклу розробки системи.

#### Глава 10. Керування конфігураціями програмних систем.

Розглядаються сучасні підходи до керування конфігурацією системи. Описано мету, завдання й способи керування, які з успіхом застосовуються багатьма колективами розробників програмних систем.

#### Глава 11. Менеджмент розробки програмного забезпечення.

Подано аналіз важливої проблеми керування проектом. Розглянуто головні причини провалів проектів, найважливіші завдання та відповідальність фахівців, котрі беруть участь у здійсненні проекту. Запропоновано систему оцінок ризиків проекту та керування ризиками.

#### Глава 12. Повторне використання у програмній інженерії.

Висвітлено підходи до використання готових напрацювань (так зване повторне використання) як до такого, що є ключовим чинником підвищення якості програмного продукту та економії потрібних для цього ресурсів, а саме часу та вартості розробки. Сформульовано основні поняття і визначення; запропоновано класифікацію кате-

горій повторно використовуваних компонент; подано підходи до створення та інформаційного забезпечення їх використання.

Глава 13. Стандарти в програмній інженерії.

Розглянуто головні чинники та напрями зазначеної діяльності.

Додаток А. Короткий огляд кодексу етики програмної інженерії.

Додаток Б. Глосарій.

Матеріал рукопису підготували: Л.П. Бабенко — глави 1—6, 8, 12, 13 та К.М. Лавріщева — глави 7, 9, 10, 11.

Книга призначена для розробників навчальних курсів вищих навчальних закладів, які готують спеціалістів з інформатики, студентів та аспірантів цих закладів, менеджерів колективів програмістів, що хочуть підвищити рівень кваліфікації (як тепер кажуть, рівень зрілості) своєї організації, фахівців проблемних галузей, які хочуть виступати в ролі грамотних замовників комп'ютерних програмних систем.

Цю роботу виконано в Інституті програмних систем НАН України в рамках проектів фундаментальних досліджень НАН України.

Автори висловлюють щирі подяку чл.-кор. НАН України П.І. Андону, який привернув нашу увагу до напрацювань в галузі ядра знань з програмної інженерії й акцентував важливість створення відповідних матеріалів, а також професорові О.Ф. Волошину за пораду створити за матеріалами лекцій, які читались у Київському університеті ім. Тараса Шевченка, навчальний посібник з програмної інженерії для студентів.

## Глава 1. ПРОГРАМНА ІНЖЕНЕРІЯ ЯК ФАХ

Виробництво й використання комп'ютерних програм є тепер масовою діяльністю: як засвідчує статистика, розробкою програм зайнято майже сім мільйонів людей, а тих, хто активно використовує програмні системи у своїй діяльності за фахом, нараховують десятки мільйонів. Програмні системи набули статусу соціально значущого фактора, який впливає на безпеку та добробут суспільства.

За таких обставин світове суспільство прийшло до висновку, що технологія виробництва програм потребує свого оформлення як самостійний інженерний фак, який має забезпечити у світі відповідний кадровий потенціал для обсягу програмних розробок, що постійно зростає. За чотири десятиріччя досвіду з програмування створено передумови для такого оформлення, і тепер ми можемо спостерігати інтенсивний процес визначення нового фаху, котрий названо програмною інженерією (Software Engineering). Цей процес протікає у двох руслах — як широка дискусія у відповідних спеціальних журналах (як, наприклад, у джерелах [1,2]) та як цілеспрямована організаційна діяльність міжнародних професійних комп'ютерних об'єднань на чолі з відомими ACM та IEEE Computer Society, для концентрації зусиль яких створено спеціальний комітет. У його публікаціях [3] дефініція програмної інженерії визначається так: **програмна інженерія** — це система методів, засобів та дисципліни планування, розробки, експлуатації й супроводження програмного забезпечення, здатна до масового відтворення.

Як бачимо, до ключових процесів у сфері діяльності даного фаху віднесено планування та супроводження. При цьому перший визначається як аналіз цілей і завдань розробки, можливості її реалізації та потрібних для цього ресурсів, а другий (супроводження) трактується не стільки як усунення знайдених хиб, скільки як визначення й реалізація необхідних змін, зумовлених еволюцією потреб та умов діяльності користувачів. Адже один з авторитетів програмної інженерії М. Джексон [4] стверджує, що золотим правилом професії є таке: всяка закінчена програмна система одразу потребує змін.

Чому мова йде саме про інженерну діяльність? Згадаймо, як вона визначається у відомому загальному тлумачному словнику [5]:

1) інженерія — це застосування наукових результатів, яке дозволяє мати користь від властивостей матеріалів та джерел енергії;

2) діяльність із створення машин для надання корисних послуг.

На відміну від науки, метою якої є здобуття знань, для інженерії знання є лише засобом отримання користі. Як казав ще один з визнаних авторитетів Ф. Брукс [6], вчений будує, щоб навчитися, інженер навчається, щоб будувати.

Розробка програмних систем визначається, як бачимо, як інженерна діяльність. Однак слід навести її досить значні відмінності від традиційної інженерії:

— традиційні гілки інженерії мають високий ступінь спеціалізації (авіоніка, машинобудування, хімія тощо). У програмній інженерії спеціалізація є помітною тільки у досить вузьких застосуваннях, як-от транслятори, операційні системи і деякі інші;

— об'єкти традиційної інженерії добре визначені, і маніпуляції з ними відбуваються у вузькому контексті, коли проблеми ухвалення рішень стосуються окремих деталей, а не загальних питань. Принципи побудови автомобіля або літака апріорно визначені, їхні типові складові — також, напрацьовано колекцію типових проектних рішень та деталей, що відповідають типовим потребам замовників: гусениці для бездоріжжя, двері позаду для транспортування поранених, шипи на шинах для ожеледиці тощо; нова розробка потребує змін лише окремих з них. У програмній інженерії подібної типізації немає;

— окремі проблеми традиційної інженерії та відповідні їм готові рішення добре класифіковані й каталогізовані. У програмній інженерії нова розробка виглядає як нова проблема, в якій досить важко розгледіти аналогії із системами, які було побудовано раніше, бо класифікації та каталогізації розв'язаних проблем практично немає.

Перелічені відмінності потребують значних зусиль для нівелювання їх, і саме тепер світова комп'ютерна спільнота визнала доцільність та вчасність таких зусиль. Це визнання матеріалізували міжнародні професійні об'єднання, створивши спеціальний комітет, метою якого є перетворення програмної інженерії на спеціальність, що визнається офіційно і має зафіксовані ознаки для розпізнавання її як такої.

Практика спеціалізації професійної діяльності, що склалася в цивілізованому світі, дозволяє вважати професію “зрілою”, якщо для неї є:

- система початкового навчання за фахом;
- механізми розвитку вмінь та навичок персоналу, необхідні для практичної діяльності;
- сертифікація персоналу, організована в рамках професії;
- ліцензування фахівців, організоване під керівництвом органів влади (зокрема, для систем з підвищеним ризиком, як-от для АЕС та їм подібних);

• системи професійного вдосконалення кваліфікації персоналу та відстеження сучасного рівня знань і технологій за фахом, щоб уможливити для фахівців виживання за умов інтенсивного розвитку фаху;

- етичний кодекс фахівців;
- професійні об'єднання.

Для програмної інженерії є лише потужні професійні об'єднання, такі, як широко відомі серед професіоналів американське об'єднання комп'ютерних спеціалістів ACM (Association for computer machinery) і комп'ютерна спілка при Інституті інженерів з електроніки та електрики IEEE Computer Society, об'єднаними зусиллями яких у 1999 році було ухвалено етичний кодекс професіонала з програмної інженерії [7].

Для наявності усіх інших з перелічених ознак потрібно, щоб було зафіксовано суму знань, які становлять фахову кваліфікацію персоналу. Згадані вище професійні об'єднання та ряд потужних виробників програмного забезпечення стали як ініціаторами, так і організаторами діяльності з визначення суми професійних знань з програмної інженерії, для чого й було створено спеціальний комітет, який об'єднав широке коло провідних спеціалістів.

У 1999 році комітет напрацював перелік головних розділів, які, на думку членів комітету, є сумою необхідних знань з даної професії, для кожного з розділів подано перелік тем, що входять до його складу [3]. Таким чином, визначено два вищі рівні дерева знань.

Назвемо ці розділи:

- аналіз вимог до програмної системи, яку має бути створено;
- детальний проект згаданої системи;
- кодування;
- тестування системи;
- процес супроводження програмного продукту;
- керування конфігурацією;
- забезпечення якості розробки;
- забезпечення відповідності розробки до вимог її замовників (валідація) та забезпечення відповідності кодів до проекту (верифікація);
- процес удосконалення отриманого програмного продукту.

Напрацьовано також перелік суміжних дисциплін, знання з яких необхідні для фахівців з програмної інженерії. До таких дисциплін, зокрема, належать:

- комп'ютерна наука;
- управління проектом;
- електрична інженерія;
- математика;
- телекомунікації та мережі;
- менеджмент;
- когнітивні науки.

Повне визначення суми знань з програмної інженерії планується отримати в 2002 році.

Отже, зміст нової інженерної дисципліни *програмна інженерія* можна вважати, до певної міри, визначеним, хоча деякі корективи цілком імовірні.

## Контрольні запитання і завдання

1. У чому суть інженерної і наукової діяльності?
2. У чому специфіка програмної інженерії як інженерної діяльності?
3. Назвіть ознаки зрілої професії. Які з них властиві програмній інженерії?
4. Які з названих ознак вимагають наявності зафіксованої суми фахових знань, що становлять основи кваліфікації персоналу.

## Література до глави 1

1. McConnel S., Tripp L. Professional Software Engineering: Fact or Fiction? // IEEE SOFTWARE. — 1999. — Nov. — Dec. — P. 13—18.
2. David L. Parnas Software Engineering Programs Are Not Computer Science Programs // IEEE SOFTWARE. — 1999. — Nov. — Dec. — P. 19—30.
3. <http://www.swebok.org.html>.
4. Jackson M. Software requirements & specifications. — Wokingham, England: Addison-Wesley, ACM Press Books, 1995. — 228 p.
5. Merriam-Webster's New Collegiate Dictionary, 10th Edition.
6. Брукс П. Мифические человеко-месяцы. — М.: Мир, 1972. — 234 с.
7. Jotterbarn D., Miller K., Rogerson S. Software Engineering Code of Ethics is Approved // Communications of the ASM. — 1999. — V. 42. — № 10. — P. 102—107.

## Глава 2. БАЗОВІ ПОНЯТТЯ ПРОГРАМНОЇ ІНЖЕНЕРІЇ

Еталонну модель програмної інженерії можна визначити як взаємодію трьох факторів:

- процесів;
- продуктів;
- ресурсів.

Кожна програмна система протягом свого існування проходить з певною послідовністю фази або стадії від задуму до його втілення в програми, експлуатацію та вилучення. Така послідовність фаз називається *життєвим циклом розробки*. На кожній фазі відбувається певна сукупність процесів, кожен з яких породжує певний продукт, використовуючи певні ресурси.

Усі продукти всіх процесів програмної інженерії являють собою певні описи — тексти вимог до розробки, погодження домовленостей, документацію, тексти програм, інструкції з експлуатації тощо.

Головними ресурсами програмної інженерії, які визначають ефективність її розробок, є час і вартість цих розробок.

Різновиди діяльності, котрі становлять процеси життєвого циклу програмної системи, зафіксовано в міжнародному стандарті ISO/IEC 12207 : 1995—0801 : Informational Technology — Software life cycle processes [1].

Згідно з наведеним стандартом, усі процеси поділено на три групи:

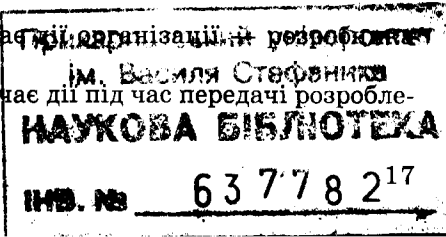
- головні процеси;
- допоміжні процеси;
- організаційні процеси.

До головних процесів віднесено такі:

— процес придбання, який ініціює життєвий цикл системи та визначає організацію-покупця автоматизованої системи, програмної системи або сервісу;

— процес розробки, який означає діяльність організації, що розробляє програмного продукту;

— процес постачання, який означає дії під час передачі розробленого продукту покупцеві;



— процес експлуатації, який означає дії з обслуговування системи під час її використання — консультації користувачів, вивчення їхніх побажань тощо;

— процес супроводження, який означає дії з керування модифікаціями, підтримки актуального стану та функціональної придатності, інсталяцію та вилучення версій програмних систем у користувача.

У свою чергу, до процесу розробки входять такі процеси:

- інженерія вимог до системи;
- проектування;
- кодування й тестування.

До допоміжних процесів віднесено ті, що так чи інакше забезпечують якість продукту. Терміном *якість продукту* позначено сукупність властивостей, які зумовлюють можливість задоволення потреб замовника, котрий сформулював їх у формі вимог на розробку.

До організаційних процесів віднесено менеджмент розробки, створення структури організації, навчання персоналу, визначення відповідальності кожного з учасників процесів життєвого циклу розробки.

Стандарт, розглянутий нами вище, є головним чинником визначення змісту діяльності у сфері програмної інженерії, і всі знання, яких потребують професіонали з програмної інженерії, формуються стосовно процесів, визначених згаданим вище стандартом. Короткий огляд стандарту див. у п. 13.

Зупинимось докладніше на процесах розробки програмного забезпечення, які в сукупності мають забезпечити шлях від усвідомлення потреб замовника до передачі йому готового продукту. На цьому шляху виділяють низку характерних робіт.

**Визначення вимог.** Збір та аналіз вимог замовника виконавцем і представлення їх у нотації, яка є зрозумілою як для замовника, так і для виконавця.

**Проектування.** Перетворення вимог до розробки в послідовність проектних рішень щодо способів реалізації вимог: формування загальної архітектури програмної системи та принципів її прив'язки до конкретного середовища функціонування; визначення детального складу модулів кожної з архітектурних компонент.

**Реалізація.** Перетворення проектних рішень на програмну систему, яка реалізує такі рішення.

**Тестування.** Перевірка кожного з модулів та способів їхньої інтеграції; тестування програмного продукту в цілому (так звана верифікація); тестування відповідності функцій працюючої програмної системи вимогам, поставленим до неї замовником (так звана валідація).

**Експлуатація та супроводження готової програмної системи.**

За десятиріччя досвіду з побудови програмних систем напрацьовано низку типових схем послідовності наведених вище робіт. Такі

схеми назвали моделями життєвого циклу. Історично першою застосовувалася так звана *водоспадна, або каскадна, модель*, за якою вважалось, що кожна з робіт виконується один раз і в тому порядку, в якому їх перелічено вище. Інакше кажучи, робилося припущення, що кожна з робіт буде виконано настільки ретельно, що після її завершення й переходу до наступної роботи повернення до попередньої не потрібно. На рис. 2.1 показано послідовність робіт за *водоспадною (каскадною) моделлю*. Повернення до початкової стадії робіт передбачається, як бачимо, лише як результат супроводження.

Цінність такої моделі полягає в тому, що вперше було зафіксовано послідовність процесів розробки та стадії готовності програмного продукту, а недоліком є те, що в її концепцію покладено модель фабрики, коли продукт проходить стадії від задуму до виробництва, після чого передається замовникові як готовий виріб, зміну якого не передбачено, хоча й можлива заміна на інший подібний продукт у разі рекламачії.

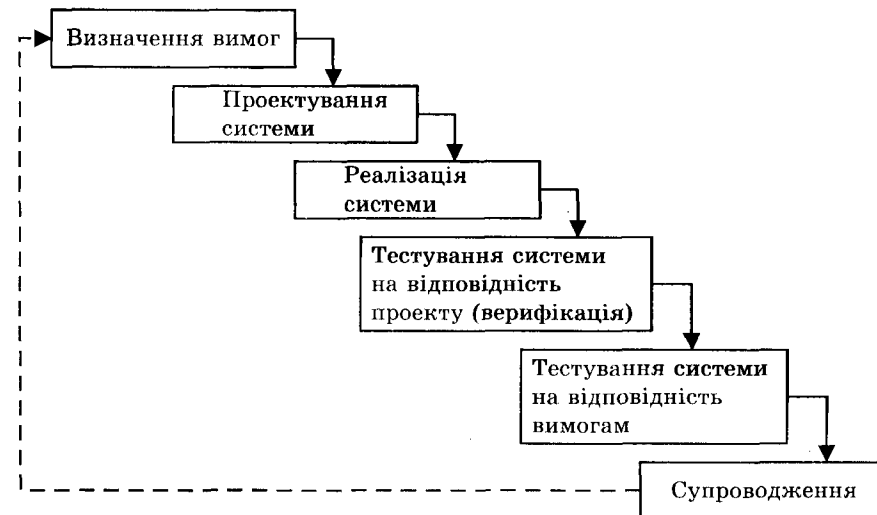


Рис. 2.1. Водоспадна модель життєвого циклу програмних систем

Для програмного продукту така модель не підходить з кількох причин. По-перше, висловлення вимог замовником — це суб'єктивний, неформалізований процес, який, як засвідчує багаторічний досвід, може багаторазово уточнюватися протягом розробки і навіть після її завершення та випробовування, якщо з'ясується, що замовник "хотів зовсім інше". По-друге, змінюються обставини та умови використання системи, тому загальновизнаним законом програмної інженерії є закон еволюції, котрий можна сформулювати так: кожна діюча програмна система з часом потребує змін або перестає використовуватися.

Зважаючи на необхідність еволюції, водоспадну модель можна розглядати як модель життєвого циклу лише для першої версії розробки. Враховуючи, що на кожній стадії робіт може виникнути потреба змін, і цю потребу має бути задоволено таким чином, щоб документація, яка є продуктом кожної стадії (опис вимог, опис проекту тощо), відповідала дійсному стану розробки після внесення змін, було створено так звану спіральну модель розвитку робіт, відміною якої є можливість багаторазового повернення до стадії формулювання вимог до розробки з будь-якої стадії робіт, якщо виявиться необхідність внесення змін.

На рис. 2.2 подано зображення спіральної моделі розробки, в якій кожний звій спіралі відповідає одній з версій розробки. На кожній стадії розробки аналізується потреба змін, а внесення змін на будь-якій стадії обов'язково починається з внесення змін до попередньо зафіксованих вимог.

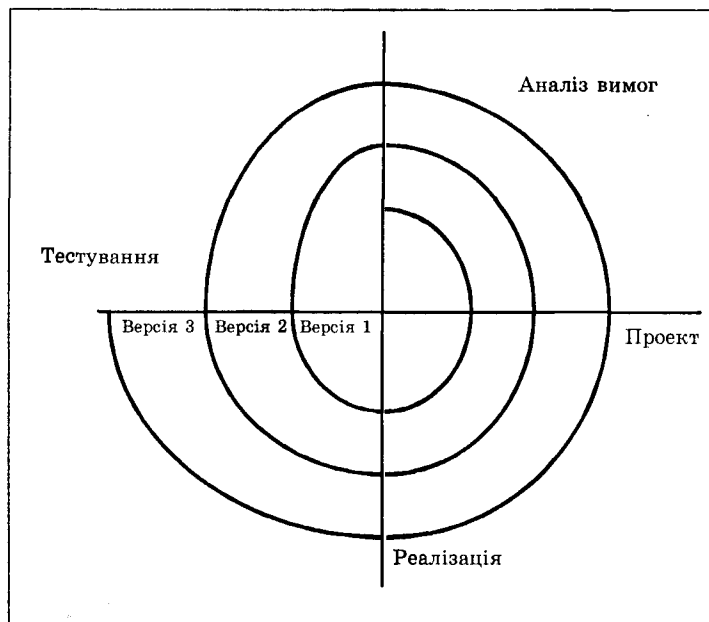


Рис. 2.2. Спіральна модель життєвого циклу розробки

## Контрольні запитання і завдання

1. Які три чинники визначають еталонну модель програмної інженерії?
2. Який вид мають продукти програмної інженерії?
3. Назвіть головні ресурси програмної інженерії, що визначають ефективність розробок програмного забезпечення.
4. Сформулюйте визначення життєвого циклу розробки програмного забезпечення.
5. Назвіть три основні групи процесів життєвого циклу.
6. Перелічіть процеси кожної з груп.
7. Який міжнародний стандарт визначає перелік і зміст процесів життєвого циклу програмного продукту?
8. Чи всі процеси, зазначені в стандарті, має бути виконано в кожній розробці програмного забезпечення або чи надає стандарт такі можливості, які можуть бути актуальними для конкретного випадку?
9. Назвіть етапи процесу розробки програмного забезпечення.
10. Основні моделі життєвого циклу розробки програмного забезпечення і їх відмінності.

## Глава 3. ІНЖЕНЕРІЯ ВИМОГ

### 3.1. Інженерія вимог як процес

Кожна система — це певний перетворювач, чію поведінку і властивості якої ми будемо в процесі створення програмної системи, і цю поведінку та властивості ми обираємо так, щоб розв'язати нашу проблему.

Програмна система — це певна машина, котра вводиться у світ для того, щоб впливати на нього. Частки світу, які впливають на машину або піддаються її впливу, становлять так званий *домен прикладної галузі* або *домен застосування*. Опис цього впливу дає відповідь на запитання “Що робить система?” і визначає вимоги до системи у формі угод між замовником і виконавцем. Як вона це робить, визначає опис машини.

Будемо називати *вимогами* до програмної системи властивості, які мають бути в системі, якщо вона адекватно виконує свої функції. Прикладами таких функцій можуть бути: автоматизація притаманних персоналу обов'язків, забезпечення керівництва інформацією, потрібною для ухвалення рішень, керування фізичними пристроями виробничого процесу тощо. Тобто, програмна система може моделювати досить складну діяльність людей та організацій, їхньої взаємодії з фізичними пристроями тощо. Відповідно до цього, вимоги до програмної системи мають відображати всі труднощі такої діяльності.

В сучасних інформаційних технологіях фаза життєвого циклу, на якій фіксуються вимоги на розробку програмного забезпечення, є визначальною для його якості, термінів та вартості робіт. Саме на цій фазі має бути зафіксовано реальні потреби користувачів, що стосуються функціональних, операційних та сервісних можливостей та які беруться реалізувати розробник. Таким чином, на цій фазі домовляються замовник і виконавець, що визначає подальші дії виконавця.

Ціна помилок та нечітких неоднозначних формулювань на цій фазі дуже висока, бо час і засоби витрачаються на непотрібну замовникові програму, оскільки він мав на увазі зовсім інше, але не зумів сформулювати свої потреби. Внесення необхідних коректив при цьому може вимагати значних переробок, а інколи й повного перепроєкту-

вання і, відповідно, перепрограмування. Тим часом статистика показує, що відсоток помилок у постановці завдань перевищує відсоток помилок кодування, і це є наслідком суб'єктивного характеру процесу формулювання вимог та майже повної відсутності засобів його формалізації. Так, згідно зі статистикою, в Америці витрачається щорічно до 82 млрд доларів на проекти, визнані після реалізації такими, що не відповідають вимогам замовника, інакше кажучи, непотрібними.

Діючими особами процесу формулювання вимог є:

- носії інтересів замовників (досить часто замовника репрезентують кілька професійних груп, потреби яких можуть мати не лише відмінності, а й навіть суперечності);

- оператори, котрі здійснюють обслуговування під час функціонування системи;

- розробники системи.

До процесу формулювання вимог входять кілька підпроцесів.

**Збирання вимог.** Джерелами відомостей про вимоги можуть бути:

- мета та завдання системи, які формулює замовник. Слід зазначити, що при безумовній пріоритетності думки замовника треба мати на увазі небезпеку неоднозначного розуміння цих формулювань замовником і розробником, а також властивість людини замовчувати багато визначальних подробиць, що є не ознакою злої волі, а лише наслідком або забудькуватості, або впевненості, що це “загально-відомо”. Тому формулювання замовника підлягають глибокому осмисленню з боку виконавця;

- діюча система або колектив, який виконує її функції. Досить часто система, котру замовляють, має замінити собою попередню систему, що вже не задовольняє замовника, або певні функції діючого персоналу. Вивчення й фіксація наявних функціональних можливостей створює базу, розширення якої дозволить сформулювати вимоги до потрібної системи, в яких врахується набутий досвід замовника. Для цього джерела також є певна небезпека перенесення вад організації попередньої системи в нову. Наприклад, розподіл обов'язків серед персоналу певного відділу склався історично впродовж послідовного розширення кадрів і є недоцільним відносно функцій відділу. Тому, вивчаючи діючу систему, треба вміло відокремити потреби проблеми, яку розв'язує система, від закладених у стару систему невдалих організаційних рішень. З огляду на це, процес урахування діючої системи під час складання вимог до нової доцільно провести за три кроки:

- а) за першим кроком вивчаємо фізичну структуру діючої системи (незалежно від того, автоматизована вона чи “людська”);

- б) за другим кроком робимо логічне узагальнення виявлених на першому етапі функцій та особливостей, виділяючи ті, що відображають

поставлену перед розробкою проблему, на відміну від закладених у стару систему окремих шляхів розв'язання проблеми (інколи не зовсім вдалих);

в) за третім кроком визначаємо логічне розширення функцій, виявлених на другому кроці, яке відповідає потребам нової системи як розвитку існуючої в заданому напрямі; визначені функції формулюємо як вимоги до нової системи;

— загальні знання щодо проблемної галузі замовника. Він формулює і розуміє свої проблеми в термінах понять певної проблемної галузі і, як згадувалося вище, замовчує подробиці, котрі належать саме до загальновідомих знань (але, на жаль, тільки серед професіоналів відповідної галузі);

— відомчі стандарти замовника, котрі стосуються організаційних вимог, середовища функціонування майбутньої системи, її виконавських та ресурсних можливостей.

Методами збирання вимог найчастіше є:

— інтерв'ю з носіями інтересів замовника та операторами;  
— спостереження за роботою діючої системи з метою відокремлення її органічних властивостей від тих, що зумовлені структурою кадрів;

— фіксація сценаріїв усіх можливих випадків використання системи, виконуваних при цьому системою функцій, ролей осіб, котрі запускають ці сценарії або взаємодіють з системою під час її функціонування.

Продуктом процесу збирання вимог є неформалізований опис цих вимог. Такий опис є фактичним контрактом на розробку між замовником і виконавцем. Обидві зазначені сторони мають розуміти його зміст, оскільки це розуміння гарантує, що система, в розробку якої буде вкладено працю виконавця, задовольнить замовника. Тому нотацію згаданого вище опису має бути орієнтовано на людину. Водночас цей опис є входом для наступного процесу інженерії вимог — аналізу вимог. Виконавцем цього процесу є розробник, а завданням є подальше уточнення й формалізація вимог та їхнє документування в нотації, що є однозначно зрозумілою колективу розробників для подальшого проектування, реалізації, тестування, документування програмного продукту та інших необхідних процесів життєвого циклу розробки (див. п. 2).

**Аналіз вимог.** Першим кроком аналізу має бути класифікація вимог. Множину зібраних вимог можна розподілити між двома головними категоріями:

— ті, що відображають можливості, які повинна забезпечити система, назвали *функціональними вимогами*;

— ті, що відображають обмеження, пов'язані з функціонуванням системи, назвали *нефункціональними вимогами*.

Перша з наведених категорій дає відповідь на запитання “Що робить система?”, а друга визначає характеристики її роботи, наприклад, що ймовірність збою системи протягом години не може перевищувати однієї мільйонної.

Нефункціональні вимоги можуть виступати як окремий чисельний показник, наприклад час чекання відповіді абонента не може перевищувати півсекунди. Інколи вони можуть мати комплексний характер і потребувати для свого втілення сукупності деталізованих властивостей, наприклад “підвищити кількість обслуговуваних клієнтів удвічі”.

Є кілька класів нефункціональних вимог, суттєвих для більшості програмних систем, які виражають обмеження, актуальні для багатьох проблемних галузей. Серед них назовемо такі:

— вимоги конфіденційності;

— відмовостійкість;

— число клієнтів, котрі одночасно мають доступ до системи;

— вимоги безпеки;

— час чекання відповіді на звернення до системи;

— виконавські якості системи (обмеження щодо ресурсів пам'яті, швидкість реакції на звернення до системи тощо).

Для більшості названих класів може бути зафіксовано спектр характерних понять, які позначаються терміном *дескриптор* і які застосовуються для розкриття їхньої суті. Склад дескрипторів закріплено у відповідних міжнародних та відомчих стандартах, що дозволяє уникнути неоднозначності тлумачення зібраних вимог.

Функціональні вимоги пов'язані із семантичними особливостями проблемної галузі, в межах якої планується розробка. Проблема термінологічних розбіжностей для них є досить впливовим фактором ускладнення. На жаль, робляться спроби розв'язання її шляхом стандартизації термінології лише для кількох проблемних галузей, для яких властивий інтенсивний розвиток комп'ютерних систем, наприклад для авіоніки та медицини. Але можна засвідчити наявність стійкої тенденції до створення стандартизованого понятійного базису більшості проблемних галузей, які набувають певного досвіду комп'ютеризації.

Наступним кроком аналізу вимог є встановлення їхньої пріоритетності, бо, як було вказано вище, вимоги, висунуті різними носіями інтересів у системі, можуть конфліктувати між собою. Крім того, кожна з вимог потребує для свого втілення певних ресурсів, надання яких може залежати також від визначеного для неї пріоритету.

Визначення впливу вимог на потреби в ресурсах є також кроком процесу аналізу вимог.

Ще одним із важливих завдань аналізу є передбачення можливих змін у зібраних вимогах і забезпечення можливостей внесення перед-

бачуваних змін без суттєвого перегляду всієї системи. Такі можливості мають забезпечити *живучість* системи та її здатність до адаптації.

Нарешті, в процесі аналізу вимог має бути перевірено правдивість та відповідність їх інтересам замовника, висловленим усіма носіями інтересів цього замовника.

Продуктом процесу аналізу є побудована модель проблеми, орієнтована на її розуміння, якого має досягнути виконавець до початку проектування системи.

## 3.2. Концептуальне моделювання проблеми

### 3.2.1. Мета

Процес побудови моделі проблеми, орієнтованої на її розуміння людиною, назовемо *концептуальним моделюванням*. Кожна проблемна галузь — назовемо її доменом — має властиву їй систему понять, знайому тільки відповідним професіоналам, свою систему “замовчування” того, що має вважатися “загальновідомим” у рамках свого домену, свої характерні властивості (атрибути), відношення та правила поведінки. Однак роль концептуальної моделі полягає в тому, щоб бути посередником між професіоналами, котрі належать до різних доменів, наприклад, як програмісти та фахівці з бухгалтерії. Ступінь формалізації цієї моделі має бути достатнім, щоб забезпечувати точність та однозначність трактування носіями інтересів у розробці, і водночас не надмірним, щоб бути доступним для розуміння не лише математикам за фахом, і спроможним відобразити деталі фахових проблем багатьох верств спеціалістів.

Враховуючи складність та розмаїття завдань, які вирішуються за допомогою програмних систем, винайти єдину модель для всіх поки що не вдалося. Але є кілька пропозицій нотацій для моделей, що досить адекватно відображають окремі аспекти проблем, а в своїй комбінації охоплюють проблеми з достатньою повнотою. Розглянемо ці пропозиції.

### 3.2.2. Онтологія домену

Найпершою з моделей домену може вважатися його понятійна база, тобто система понять, за допомогою якої формулюються всі аспекти проблеми. Понятійна база визначає не лише термінологію, якою мають користуватися носії інтересів, котрі беруть участь у процесі аналізу вимог, а також і суттєві відносини між поняттями та їхньою інтерпретацією.

Сукупність термінології, понять, характерних для них відносин і парадигми їхньої інтерпретації в межах домену прийнято називати *онтологією домену*.

Зміст поняття може бути охарактеризовано сукупністю спільних істотних ознак тих явищ та предметів, які позначаються найменням *поняття*. Сукупність явищ, охоплених поняттям, називається його *обсягом*. У ментальній діяльності людини застосовуються відношення, які дозволяють будувати похідні поняття і встановлювати між ними зв'язки. Серед таких відношень назовемо найпоширеніші:

— *узагальнення* — це звуження істотних ознак поняття, при цьому розширюється коло охоплених поняттям об'єктів, тобто його обсяг. Приклади: свійська тварина є узагальненням понять свиня, корова та багатьох інших;

— *конкретизація* — це додавання істотних ознак, завдяки чому зміст поняття розширюється, а обсяг поняття звужується. Наприклад, студент консерваторії є конкретизацією загального поняття студент;

— *агрегація* — це об'єднання низки понять у нове поняття, істотні ознаки нового поняття при цьому можуть бути або сумою ознак компонент або суттєво новими. Прикладом першого типу є пральна машина з центрифугою, другого — автомобіль як композиція колес, двигуна, корпусу, керма тощо. Відношення агрегації часто має назву відношення частка — ціле;

— *асоціація* — це найбільш загальне відношення, що утверджує наявність зв'язку між поняттями, не уточнюючи залежності між їхнім змістом та обсягами.

Для окремих доменів можуть використовуватися специфічні для них відношення. Наприклад, каталізатор — у хімії, нащадок-предок — у генеалогії, рід-вид — у біології тощо. Фіксація в онтології відношень між поняттями структурує понятійний базис домену проблемної галузі.

Онтологія дозволяє втримувати користувача в максимально можливому просторі визначених наперед можливостей, зміст яких зафіксовано і він зрозумілий як розробникові, так і замовникові. Очевидно, що перебування в такому просторі гарантує обидві згадані вище сторони договору (який матеріалізовано у формі вимог) від взаємних непорозумінь. Очевидно також, що для відносно нових і достатньо складних галузей застосування комп'ютерних систем повний спектр можливостей передбачити важко, тому важливу роль відіграє аспект внесення змін у вимоги.

Для представлення онтологій використовуються спеціальні нотації. Наприклад, діаграми сутність—зв'язок мають форму графів, у вузлах яких лежать поняття, а дуги відповідають відношенням між ними. Слід зазначити, що у зв'язку з інформатизацією життя суспільства поширюється тенденція створення і стандартизації онтологій для окремих проблемних галузей, що суттєво спрощує формулювання вимог у таких галузях.

У розробку галузевих онтологій вкладається чимало інвестицій, навіть побудовано бібліотеки фахових онтологій [3], та, на жаль, не в Україні, де така робота ще й не починалася. Треба сказати, що побу-

дова й супроводження національних фахових онтологій та гармонізація їх із закордонними зразками принесе велику віддачу від вкладених інвестицій, дозволить занотувати актуальний стан справ у відповідній галузі, створити середовище взаєморозуміння й обміну думками вітчизняних професіоналів як між собою, так і з міжнародною спільнотою відповідних спеціалістів.

У значущих для суспільства професіях така робота лежить у руслі помітної тенденції фіксації необхідної суми професійних знань, що дозволяє проводити акредитацію відповідних навчальних програм навчальних закладів та сертифікацію спеціалістів як у масштабі окремої держави, так і в міжнародному масштабі. Отже, усе зазначене засвідчує, що розгортання робіт із створення національних онтологій уже на часі і є кроком до інтеграції наших фахівців у міжнародний контекст.

### 3.2.3. Моделі динамічних явищ домену

Розглянуті в п. 3.2.2 онтології відображають статичні властивості явищ домену, які не враховують їхніх змін за часом. Але більшість проблем, котрі розв'язують за допомогою комп'ютерів, мають справу з динамічними явищами, для яких відстеження їхньої поведінки з перебігом часу є суттєвим аспектом вимог до системи.

Домен з такими властивостями назвали динамічними доменами.

Для динамічних доменів суттєвими поняттями є:

- *стан* (домену, системи, об'єкта тощо) — фіксація певних властивостей на певний момент або інтервал часу;

- *інтервал стабільності* — інтервал часу, протягом якого не змінюється стан;

- *подія* — явище, що провокує зміну певного стану як перехід до іншого стану (ми розглядаємо лише дискретні процеси в доменах).

Поведінка домену в часі розглядається як прогрес від стану до стану. Серед динамічних доменів визначаються такі різновиди:

- *інертні домени*, зміна станів яких ніколи не відбувається з ініціативи об'єктів домену, а реалізується під керівництвом зовнішніх агентів. Наприклад, обробка текстів реалізується як результат виконання послідовних команд оператора;

- *реактивні домени*, зміна стану в яких є відповіддю на певну зовнішню подію. Наприклад, автомат, який видає цукерку у відповідь на монетку, кинуту в щілину монетоприймача;

- *активні домени*, які можуть переходити від стану до стану без зовнішніх стимулів, як це робиться з людиною або атмосферою.

Визначаючи вимоги до системи, важливо одразу вирішити, чи планується вона як інертна, реактивна, чи як активна. Наприклад, для медичної системи принципово визначити, чи планується вона як модель пацієнта, чи як реєстратор його стану.

Найпоширенішою моделлю поведінки динамічних явищ є так звана модель переходів у стани, яка базується на моделі скінченного автомата. Для кожного явища або об'єкта, котрий має динамічну поведінку, фіксуються стани, в яких він може перебувати. Дозволяється мати кінцеву множину станів, при чому перехід з одного стану до іншого відбувається дискретно і зумовлюється тим, що відбулася певна подія. Модель поведінки визначає стани, властиві домену, події, які впливають на зміну стану в домені, та послідовність зміни станів залежно від послідовності подій, що відбуваються. Наприклад:

- якщо поїзд має стан “стоїть на станції”, і диспетчер станції дав сигнал відправки, тобто відбулася подія “сигнал відправки”, поїзд переходить у стан “початок руху від станції”;

- якщо хворий має стан “приймання процедури електрофорезу”, і відбулася подія “дзвоник годинника задзвенів”, медик має перевести хворого у стан “процедуру прийняв”.

Якщо змінюється стан якогось об'єкта, то це найчастіше супроводжується певною послідовністю дій, які треба при цьому виконати. Тому модель поведінки зазвичай має у своєму складі визначення сукупностей дій, котрі потрібно виконувати при переході до відповідних станів. При викладенні конкретних методів аналізу вимог буде наведено приклади відповідних нотацій для представлення моделі переходів у стани.

### 3.2.4. Модель алгоритмів

Моделі алгоритмів, які застосовуються для формулювання вимог, мають визначити ті дії або процеси, котрі супроводжують переходи з одного стану в інший.

Зазвичай такі моделі визначають потоки даних, які передаються від стану до стану чи від об'єкта до об'єкта, та потоки управління між ними.

## 3.3. Об'єктно-орієнтована інженерія вимог

Кожна з наведених у п. 3.2 моделей так чи інакше має на меті перебороти складність розв'язуваної проблеми шляхом її декомпозиції на окремі простіші та зрозуміліші складові. Досі ми не обговорювали, що таке складові. Сутностями, для яких визначаються стани, можуть бути фізичні об'єкти (людина, двигун), явища природи (повінь, атмосфера), політичні явища (вибори, мітинг), функціональні блоки системи керування (відділ кадрів, відділ збуту) тощо. Між тим, від того, яким чином ми будемо структурувати проблему, від нашого бачення її складових і взаємодії (інтерфейсів) цих складових залежить зрозумілість та “прозорість” опису тих вимог, який буде результатом процесу аналізу, його повнота й точність.

Типи складових компонент та правила їхньої композиції (інтерфейси) визначаються в програмній інженерії як *архітектура системи*. Модель декомпозиції проблеми, яка визначає архітектуру, називається *парадигмою програмування*. Відомими парадигмами, поширеними в програмній інженерії, є дві: модель функції-дані та об'єктна.

*Модель функції-дані* є історично першою. Згідно з нею, проблема декомпонується на послідовність функцій та даних, які обробляються за допомогою цих функцій. Тобто, елементами композиції є дані та функції над ними. Представлення перших і других має бути узгоджено між ними. Якщо змінюються якісь дані, треба переглянути всі функції, які їх обробляють, і визначити, чи не потребують деякі з них змін також. Якщо змінюється функція, треба переглянути всі дані, які вона обробляє чи має своїм результатом, з метою пошуку тих, котрі залежать від внесених змін. Можна зробити висновок, що за такою парадигмою внесення локальних змін до постановки проблеми потребує ревізії всіх даних та всіх функцій, щоб бути певним, що вони не зазнали впливу внесених змін.

*Парадигма об'єктно-орієнтованого підходу* до розробки програмних систем дозволяє уникнути зазначеного вище недоліку. Згідно з її концепцією загальне бачення проблем пропонується як таке, що узгоджується з викладеними нижче постулатами:

- світ становлять об'єкти, які взаємодіють між собою;
- кожний об'єкт має певний набір властивостей або атрибутів (аналог суттєвих ознак поняття);
- атрибут визначається своїм іменем та значеннями, які він може приймати;
- об'єкти можуть бути у відношеннях одне з одним;
- значення атрибутів та відношення можуть із часом змінюватись;
- сукупність значень атрибутів конкретного об'єкта в певний момент часу визначає його стан;
- сукупність станів об'єктів визначає стан світу;
- світ та його об'єкти можуть перебувати в різних станах;
- у певні інтервали часу можуть виникати якісь події;
- події можуть спричиняти інші події або зміни станів;
- протягом часу кожний об'єкт може брати участь у певних процесах, які зводяться до виконання послідовності дій, різновидами котрих є переходи з одного свого стану в інший під впливом відповідних подій, збудження певних подій чи посилка певних повідомлень до інших об'єктів;
- дії, які можуть виконувати об'єкти, називають *операціями об'єктів* (як синоніми використовують також терміни *методи об'єкта* та *функції об'єкта*);

- можливі сукупності дій об'єкта називають його поведінкою;
- об'єкти взаємодіють шляхом обміну повідомленнями;
- об'єкти можуть складатися із частин.

Об'єкт — це певна абстракція даних та поведінки: множина екземплярів із спільним складом атрибутів та спільною поведінкою становить клас об'єктів. Визначення класу проведено за так званим *принципом приховування інформації*, котрий можна сформулювати так: повідомляйте користувачеві лише ті відомості, які йому потрібні для того, щоб скористатися вашими напрацюваннями, все інше приховуйте. Такий принцип має низку переваг, серед яких суттєві такі:

- користувача позбавлено необхідності знати зайве, тобто непотрібне;
- того, про що йому не повідомили, він не може пошкодити, тобто здійснено захист від його неправомірних дій, як навмисних, так і випадкових;
- усе, про що не знає користувач, можна змінювати, і це не справлятиме на нього ніякого впливу.

Визначення об'єктів проведено згідно з наведеним принципом і складається воно з двох частин — видимої та невидимої. Перша з них містить усі відомості, які потрібні для того, щоб взаємодіяти з об'єктом, і називається *інтерфейсом об'єкта*, а друга містить подробиці його внутрішнього устрою, і вони сховані, або, як кажуть, інкапсульовані (тобто перебувають немовби в капсулі). Так, наприклад, якщо нашим об'єктом є прилад, котрий реєструє показники температури, то до видимої частини його визначення відносимо операцію показу актуального значення температури. Якщо ж до того бажано керувати гранично дозволеними діапазонами температур, як у тепличному господарстві, то ці діапазони має бути визначено як видимі, тобто віднесено до інтерфейсу об'єкта. Іншим прикладом об'єкта може бути банк, зовнішня поведінка котрого виглядає як можливість виконувати “видимі” операції відкриття й закриття рахунка клієнта, поповнення чи зменшення рахунка клієнта, тоді як усі внутрішні дії служб банку з обслуговування рахунка, що забезпечують виконання “видимих” операцій, а також усі реєстри та інші внутрішні дані банку щодо клієнтів і наявності готівки в касирів інкапсульовано, тобто клієнтові про них знати непотрібно і неможливо.

Ще одним важливим засобом визначення об'єктів є так зване *успадкування*. Кажуть, що один клас об'єктів успадковує інший, якщо він повністю містить усі атрибути й поведінку успадкованого класу, але має ще додаткові атрибути та (або) поведінку. Клас, який успадковує, називається *суперкласом*, а клас, що успадковує, називається *підкласом*. Спадковість явним способом фіксує спільні та розбіжні риси об'єктів і дозволяє явно виділити складові компоненти проблеми, які можна використати в кількох випадках

шляхом побудови для них декількох класів-спадкоємців. Класи можуть утворювати ієрархії спадкоємців довільної глибини, в котрих кожний відповідає певному рівню абстракції і є узагальненням класу-спадкоємця і конкретизацією класу, який успадковує сам. Наприклад, клас числа може мати спадкоємців — підкласи цілі числа, комплексні числа, дійсні числа. Усі наведені підкласи успадковують операції суперкласу (відомі арифметичні операції складання та віднімання), але кожний з підкласів визначає свої особливості виконання згаданих операцій.

Відзначимо одну з найважливіших властивостей операцій об'єкта, яку названо *поліморфізмом операцій*. Сутність її полягає в тому, що об'єкт, котрий відправляє повідомлення, не може знати класу того об'єкта, який отримає відіслане повідомлення. Інтерпретація ж отриманого повідомлення визначається класом того, хто його одержить. Наприклад, повідомлення об'єктові А виконати операцію додавання до В буде трактуватись як додавання матриць, якщо класи об'єктів А та В визначено як матриці, або як додавання цілих, якщо А та В належать до класу цілих.

Завдяки багатим можливостям абстракції даних і поведінки, інкапсуляції та можливостям успадкування об'єктно-орієнтований підхід до розробки програм набув великої популярності і фактично витіснив усі інші. У п. 2 ми вказували, що модель діяльності з програмної інженерії визначається за допомогою трьох факторів — процесів, продуктів та ресурсів; послідовність процесів визначає стадії життєвого циклу розробки програмних систем, і продуктами кожного з процесів є описи, які відповідають моделі певної стадії осмислення проблеми. За об'єктно-орієнтованим підходом усі зазначені вище моделі мають визначатися як взаємодія певних об'єктів, при цьому в моделі вимог фігурують об'єкти, взаємодія котрих визначає проблему, яку маємо розв'язувати за допомогою програмної системи, а в подальших моделях (у моделі проекту, моделях реалізації та тестування) йдеться про об'єкти, взаємодія яких визначає розв'язання тієї проблеми (моделі проекту та реалізації) або перевірку достовірності того розв'язання (модель тестування).

Відповідно до зазначеного вище, у світі запропоновано чимало методів об'єктно-орієнтованого аналізу вимог, об'єктно-орієнтованого проектування програм, об'єктно-орієнтованого програмування (згадаймо широко поширений C++). Але найбільшу цінність серед них мають ті, які узгоджені між собою.

Якщо вдається встановити відповідність між об'єктами моделей продуктів різних стадій (процесів) життєвого циклу розробки, кажуть, що моделі дозволяють *трасування вимог*. Трасування вимог — це можливість простежити послідовну трансформацію об'єктів вимог, зрозумілих замовникові, у відповідні компоненти продуктів

послідовних стадій розробки, закінчуючи діючою програмною системою. Можливість трасування полегшує контроль за здійсненими трансформаціями та за внесенням змін протягом усього циклу розробки синхронно в усі напрацьовані продукти різних стадій розробки до її завершення, що відповідає спіральній моделі життєвого циклу (див. п. 2).

За парадигмою об'єктно-орієнтованого підходу концептуальне моделювання проблеми (дивись п. 3.2) відбувається в термінах взаємодії об'єктів:

- онтологія домену визначає склад об'єктів домену, їхніх атрибутів та взаємовідношень, а також послуг (операцій);

- модель поведінки визначає можливі стани об'єктів, інциденти, які ініціюють переходи з одного стану в інший, повідомлення, які об'єкти надсилають одне одному;

- модель процесів визначає дії, які виконують об'єкти.

Всі успішні пропозиції об'єктних методів, а їх напрацьовано вже чимало, мають у своєму складі зазначені вище моделі, хоча вони й відрізняються своїми нотаціями та деякими іншими деталями. Одразу зазначимо, що жодна з них ще не набула загального визнання, при тому що багато з них мають досить широке коло прихильників і позитивний досвід застосування. Нижче ми зупинимося більш детально на трьох методах.

Метод С. Шлеєр та С. Меллора (див. п. 3.2.2) ми обрали переважно тому, що читач може з вітчизняної публікації [2] отримати досить повне уявлення про його можливості, які є, до певної міри, типовими.

Метод сценаріїв використання майбутньої системи, або сценарний підхід (див. п. 3.2.3), є, на погляд авторів, єдиним методом, який вказує шлях до виявлення об'єктів, суттєвих для домену проблемної галузі. Справді, всі методи декларують — як перший крок — виявлення об'єктів і попереджають, що вдалий склад об'єктів зумовить зрозумілість і точність вимог, але тільки сценарний підхід дає рекомендації щодо того, з чого починати пошук об'єктів.

Метод UML (Unified Modelling Language) є узагальненням цих двох методів та кількох інших і тепер претендує на те, щоб стати міжнародним стандартом методу аналізу вимог та проектування програмних систем.

### 3.4. Метод інженерії вимог С. Шлеєр та С. Меллора

Програмна система розглядається як сукупність визначеного ряду доменів проблемних галузей, кожний з яких є окремим світом, населеним своїми об'єктами, і котрий аналізується незалежно від інших. Продуктом аналізу домену є три складові, які будуються, відповідно, за три етапи, а саме:

- онтологія домену, яку автори даного методу називають інформаційною моделлю системи або інформаційною моделлю домену;
- модель станів об'єктів, визначених у складі інформаційної моделі (або онтології);
- модель процесів, які супроводжують переходи з одного стану в інший.

#### 3.4.1. Інформаційна модель або онтологія домену

**Пошук об'єктів.** Як ми вже зазначили в п. 3.2.2, завданням першого етапу є виявлення суттєвих об'єктів і встановлення зв'язків (відношень) між ними. Для цього ми маємо, по-перше, знайти такі об'єкти, а по-друге, дати їм унікальні та значущі назви. Ці дії суто суб'єктивні, єдина рекомендація щодо цього, яку надали автори методу, полягає в переліку категорій, серед яких доцільно проводити пошук. Це такі категорії:

- реальні предмети світу, котрі фізично втілені, як, наприклад, Іван Іванович, стілець у кабінеті, Дніпро;
- абстракції фізичних предметів світу, як, наприклад, людина, тварина, літак, кабель, дім, сад;
- ролі предметів, тобто абстракції їхнього призначення або мети використання. Наприклад, для домену університет значущими є ролі декан, ректор, студент, викладач; для домену хімічне виробництво — ролі очисна споруда, каталізатор, відстійник; для домену адміністрація міста — платник податків, виборець, мер;
- інциденти, тобто абстрактні події, котрі впливають на зміну стану об'єкта, як, наприклад, паводок, вибори, залік, прогул;
- взаємодії, тобто об'єкти, які характеризують відношення об'єктів, як, наприклад, контракт, перехрестя доріг або вулиць, угода;
- специфікації, тобто представлення правил, стандартів, критеріїв якості, обмежень користування.

Тож почнемо аналіз домену проблемної галузі послідовно для кожної з перелічених вище категорій явищ, виявляючи суттєві об'єкти та їхні класи (див. п. 3.3.1).

Для класів об'єктів вибираються значущі імена, унікальні в межах домену.

**Атрибути об'єктів.** Склавши список виявлених об'єктів, для кожного з них потрібно визначити його характерні ознаки або властивості,

які в інформатиці називають атрибутами. Кожний атрибут є абстракцією однієї з характеристик об'єкта, котрі властиві всім представникам класу об'єктів. За атрибутом закріплюємо ім'я, унікальне в межах класу.

Зазначимо, що вдало вибрані імена (як кажуть, мнемонічні імена, тобто такі, котрі передають сутність об'єктів, які вони позначають) є важливим чинником для розуміння програм.

Для кожного з визначених атрибутів задаються його можливі значення (типи значень). Способи опису можливих значень можуть бути такі:

- числовий діапазон;
- перелік можливих значень;
- посилання на документ, у якому визначено можливі значення;
- правила генерації значень.

**Ідентифікатори об'єктів.** Для об'єкта визначається так званий ідентифікатор — це один або кілька атрибутів, значення чи сукупність значень яких точно вирізняють екземпляр об'єкта з-поміж інших у класі. Прикладом ідентифікатора можуть бути: назва або ім'я об'єкта, табельний номер працівника (бо серед них можливі носії однакових прізвищ та імен), номер паспорта, код платника податків, номер автомобіля. Сукупність атрибутів, які становлять ідентифікатор, може залежати від області визначення об'єкта. Так, якщо йдеться про котів однієї родини, то зазвичай кличка kota є унікальною у межах родини, якщо ж йдеться про котів одного двору, то, можливо, доведеться уточнити кличку kota його мастю (Васько.рудий) або ім'ям його хазяїна (Васько.Олена).

Посилання на ідентифікатор подається як перелік через крапку атрибутів, котрі входять до складу ідентифікатора, як це видно з наведених вище прикладів. Так само посилання на атрибут при необхідності може уточнюватися класом, поданим через крапку, як, наприклад: викладач.стаж-роботи, літак.розмах-крил, собака.порода.

Представлення інформаційної моделі в цьому методі базується на відомій реляційній моделі даних. Атрибути об'єктів представляються як атрибути відношень за такими правилами:

- кожний екземпляр об'єкта одночасно обов'язково має одне значення (тобто значення не може бути відсутнім або невизначеним);
- атрибут є одновимірним і не має внутрішньої структури або кількох значень одночасно;
- якщо до ідентифікатора входить кілька імен атрибутів, усі вказані імена атрибутів, окрім першого, належать до першого вказаного імені, яким є ім'я об'єкта.

Об'єкти нумеруються. Об'єкт зображається прямокутною рамкою, всередині якої подається номер та ім'я об'єкта, а також імена його атрибутів, як, наприклад, на рис. 3.1:

<b>1. Собака</b> <ul style="list-style-type: none"> <li>* Ім'я</li> <li>* Порода</li> <li>* Стать</li> <li>* Вік</li> </ul>	<b>2. Адреса</b> <ul style="list-style-type: none"> <li>* Вулиця</li> <li>* Номер будинку</li> <li>* Номер квартири</li> </ul>	<b>3. Особа</b> <ul style="list-style-type: none"> <li>* Прізвище</li> <li>* Ім'я</li> <li>* По батькові</li> <li>* Номер паспорта</li> </ul>
---	--	---

Рис. 3.1. Зображення об'єктів з атрибутами

**Зв'язки об'єктів.** Визначивши склад класів об'єктів домену та властиві їм атрибути, розглянемо зв'язки (відношення) між об'єктами домену.

Об'єкти одного класу можуть брати участь у бінарних, тобто парних зв'язках з об'єктами іншого або того самого класу.

Розглянемо кілька прикладів зв'язку:

- 1) власник авто володіє авто, а авто належить власникові;
- 2) прибиральник прибирає кімнату, а кімната прибирається ним;
- 3) проект ведеться керівником, керівник веде проект;
- 4) керівник займає кімнату, кімната належить керівникові;
- 5) літак займає доріжку аеродрому, доріжка зайнята літаком;
- 6) проект має виконавців, виконавці зайняті в проекті;
- 7) керівник керує виконавцем, виконавець підпорядкований керівникові;

- 8) виконавець займає кімнату, кімната містить виконавця.

Кожна фраза фіксує можливість екземпляра певного класу об'єктів бути у певному відношенні (або зв'язку) з екземпляром іншого класу (приклади 1—5), або того самого класу (приклад 6). Суттєвою рисою наведених зв'язків є число екземплярів об'єктів, які можуть одночасно брати в ньому участь.

Розрізняються три фундаментальних види зв'язку:

- один до одного ( $1 : 1$ ), коли у зв'язку беруть участь по одному екземпляру з кожного боку (приклад 5). Одночасно на одній злітній доріжці може бути тільки один літак, і один літак може займати тільки одну доріжку. Приклади 3 та 4 можуть також бути прикладами зв'язку  $1 : 1$ , якщо в організації керівник займає окремий кабінет і керує одноосібно тільки одним проектом;

- один до багатьох ( $1 : n$ ), коли один екземпляр об'єкта певного класу може підтримувати відношення одночасно з декількома екземплярами об'єктів іншого або того самого класу. Приклад 1. Якщо виключається спільне володіння авто, але можна володіти кількома машинами. Інший приклад — 6, коли керівник може мати декілька

підлеглих, але в кожного з них один шеф. Якщо керівник може керувати кількома проектами, то приклад 4 також належить до такого виду зв'язку;

- багато до багатьох ( $m : n$ ), коли у зв'язку можуть брати участь по декілька екземплярів об'єктів з кожного боку. Це ілюструють приклади 1, якщо тими самими кількома автомашинами дозволяється володіти декільком особам, та 2, коли декілька прибиральників прибирають по черзі кілька кімнат.

Ми бачимо, що опис зв'язків відображає певні вимоги до статичних залежностей, які бувають для задач, котрі має розв'язувати програмна система. Метод С. Шлеєра та С. Меллора передбачає спеціальну графічну нотацію для фіксації зв'язків, що базується на діаграмах відомої моделі Чена сутність — зв'язок (entity — relations) і представляє інформаційну модель (онтологію) проблемної галузі.

Представлення інформаційної моделі провадиться в такий спосіб.

Зв'язки між об'єктами зображаються стрілками, що вказують напрямок зв'язку. Біля рамки об'єкта, котрий бере участь у зв'язку, на лінії стрілки вказується роль, яку цей об'єкт підтримує в даному зв'язку. Зв'язок  $1 : 1$  позначається двоспрямованою стрілкою, що має по одному “наконечнику” стрілки з кожного боку. Зв'язок  $1 : n$  позначається стрілкою, що має два “наконечники” з боку того об'єкта, для якого у зв'язку можуть брати участь декілька екземплярів, і, нарешті, по два “наконечники” з кожного боку має стрілка, яка позначає зв'язок виду  $n : m$ .

Над стрілкою може вказуватися назва (ім'я) зв'язку. Зв'язки, наведені вище, є безумовними, бо обов'язково кожний екземпляр об'єкта заданого класу бере участь у вказаному зв'язку. Передбачено також можливість умовних зв'язків, коли окремі екземпляри об'єктів певного класу за певних умов можуть не брати участі у зв'язку, в цьому разі відповідний кінець стрілки позначається літерою *y*. Наприклад, деякі доріжки аеродрому в певний момент часу можуть бути вільні від літаків. На рис. 3.2 представлено фрагменти інформаційної моделі, кожний з яких відповідає одному з прикладів зв'язків 1—8, наведених вище.

При цьому за назву зв'язку обрано літеру *R*, за якою стоїть номер прикладу.

Інформаційна модель проблеми на наступних фазах життєвого циклу розробки програмної системи відображається на структури баз даних. Власне, таке відображення є продуктом проектних рішень з реалізації зв'язків, задекларованих як вимоги до розробки, що буде розглянуто у п. 4.2.1.

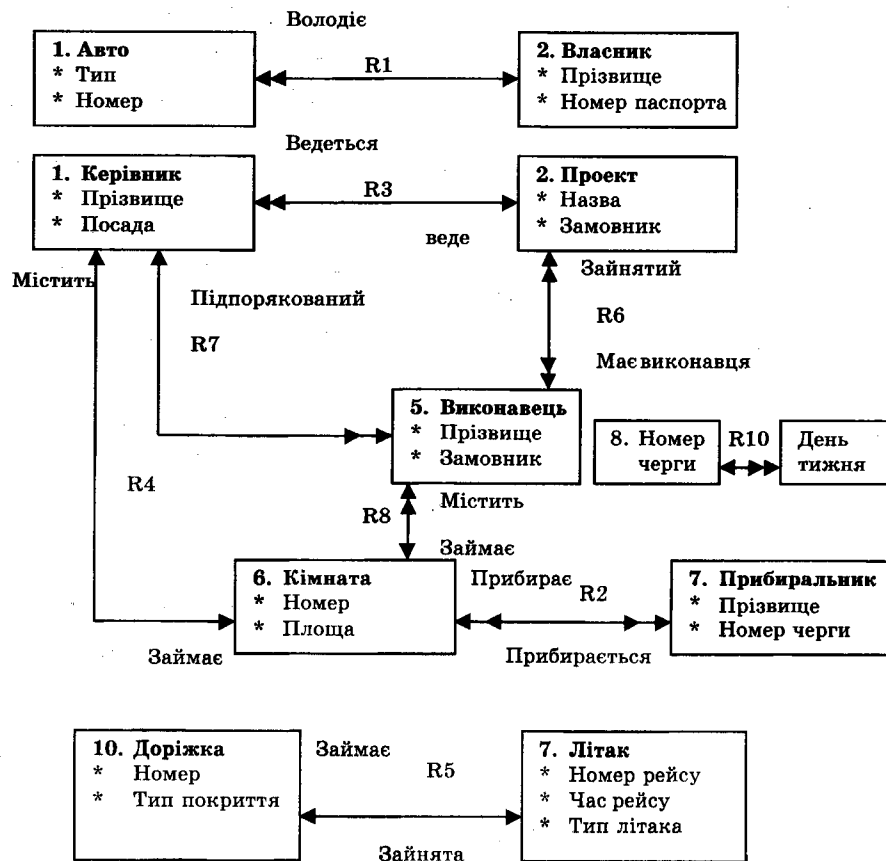


Рис. 3.2. Приклади фрагментів інформаційної моделі

Є одне відношення, яке має особливу вагу для представлення онтологій. Це відношення успадкування (див. п. 3.3.1), за допомогою якого виражаються спільності та розбіжності між визначеними класами об'єктів. Зазвичай, відношення успадкування подаються на окремих діаграмах — на так званих діаграмах класів. На рис. 3.3 наведено приклади таких діаграм.

При цьому діаграму інформаційної моделі супроводжують неформальним описом усіх об'єктів, їхніх атрибутів та зв'язків, в яких об'єкти беруть участь.

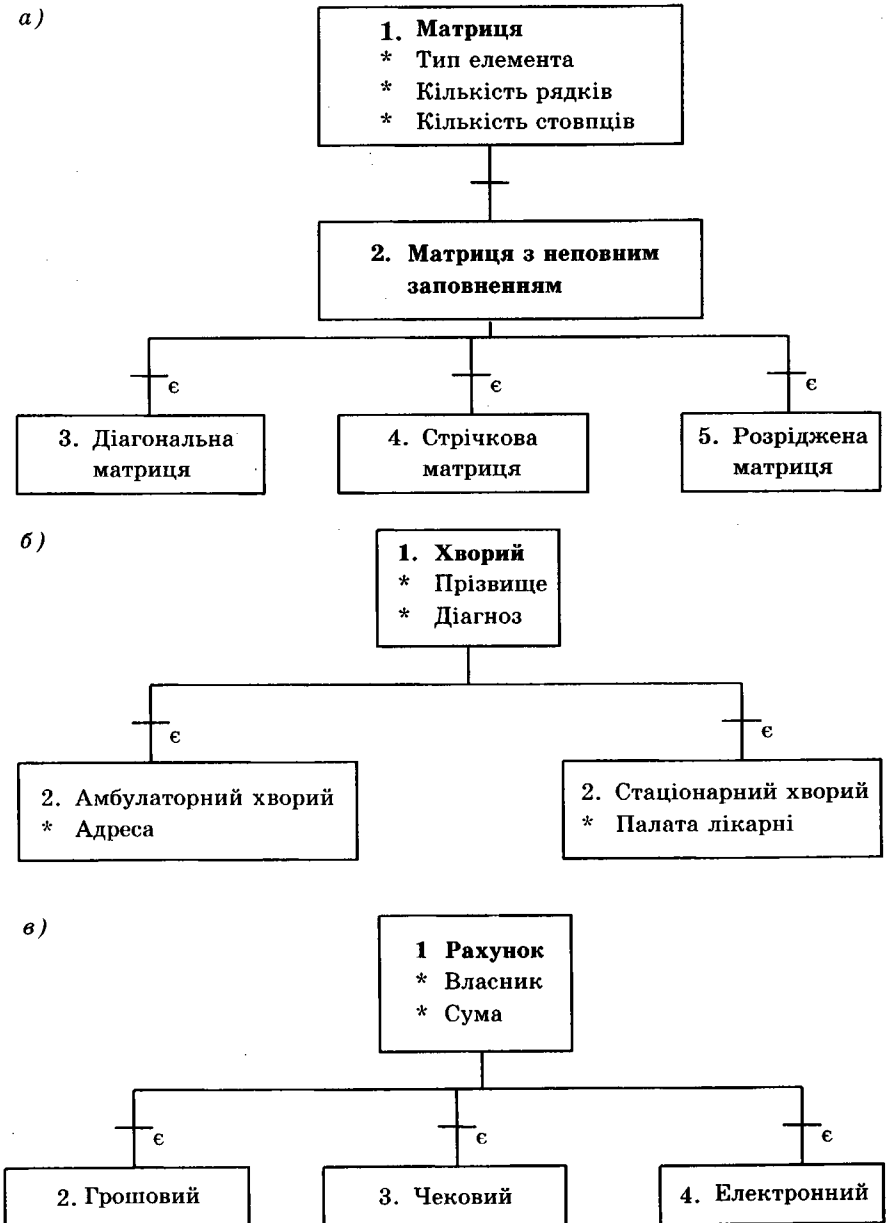


Рис. 3.3. Приклади діаграм класів

### 3.4.2. Модель станів

Модель, яку буде розглянуто нижче, має відображати динаміку змін, котрі відбуваються в стані кожного з визначених у п. 3.4.1 класів об'єктів, або, як кажуть, динаміку їхньої поведінки. Зазначимо, що всі екземпляри одного класу об'єкта, за визначенням поняття клас, мають однакову поведінку. Нагадаємо наведені у п. 3.2.3 базові поняття моделі динаміки поведінки об'єктів:

- стан об'єкта визначається поточними значеннями окремих його атрибутів, а стан домену — сукупністю станів його об'єктів;
- стан об'єкта змінюється внаслідок того, що відбулися певні події або з'явилися певні стимули;
- зміна стану супроводжується певними процесами, котрі визначено для кожного стану як такі, що мають відбутися після досягнення цього стану.

У методі, який ми розглядаємо, запропоновано одразу дві альтернативні нотації для фіксації динамічних аспектів вимог як поведінки визначених об'єктів. Одна з них — графічна — називається *діаграмою переходів у стани* (ДПС). Друга — таблична — називається *таблицею переходів у стани* (ТПС). Обидві нотації базуються на автоматі Мура, відомому з теорії автоматів. Згідно з цим методом, побудову моделі станів починаємо з того, що серед визначених інформаційною моделлю класів об'єктів виділяємо ті, котрі мають динамічну поведінку, тобто змінюють свій стан з плином часу, або, як кажуть, мають життєвий цикл від створення екземпляра об'єкта через зміни його станів і до зникнення об'єкта. Для відображення поведінки таких об'єктів у вимогах до розробки потрібно:

- 1) визначити множину станів, в яких об'єкт може перебувати, при тому кожний стан є абстракцією стану, в котрому може перебувати кожен з екземплярів класу об'єктів;
- 2) визначити множину інцидентів або подій, які спонукають екземпляри класу змінювати свій стан;
- 3) визначити для кожного із зафіксованих станів правила переходу, котрі вказують, в який новий стан перейде екземпляр даного класу, якщо певна подія з визначеної для класу множини подій відбудеться тоді, коли він перебуває в даному стані;
- 4) визначити для кожного з визначених станів дії або процеси, які потрібно виконати при набутті даного стану.

Графічна нотація для представлення наведеної вище інформації передбачає таке:

- кожному стану, визначеному для класу об'єктів, присвоюють назву та порядковий номер;
- кожній визначеній події присвоюють унікальну мітку та назву;
- на діаграмі ДПС стан позначається рамкою, яка містить номер та назву стану;

- перехід від стану до стану зображається спрямованою дугою, позначеною міткою та назвою події, яка зумовила перехід;

- початковий стан позначається стрілкою, що веде до відповідної йому рамки, і є станом, в якому екземпляр об'єкта з'являється вперше (ініціалізується). Допускається кілька початкових станів на ДПС;

- заключний стан визначає кінець життєвого циклу екземпляра об'єкта, що може настати в одному із двох випадків — або екземпляр існує далі, але його поведінка втрачає динамічний характер, і тоді такий стан позбавляється спеціальної позначки, або екземпляр зникає, і тоді заключний стан позначається пунктирною рамкою;

- під рамкою зазначаються дії, які має виконати екземпляр об'єкта, коли він набуває відповідного рамці стану.

Зазначимо, що при застосуванні ТПС дії, відповідні станам, позначаються окремою нотацією.

Зупинимось коротко на можливих діях при зміні станів.

Дії виконуються екземпляром, котрий змінює стан.

Подія, яка викликає зміни стану, є сигналом керування, що зазвичай передає якісь дані. Вони мають нести досить інформації, щоб визначити екземпляр класу, котрий змінює стан (або створити новий екземпляр) і забезпечити даними відповідні дії. Різновиди дій такі:

- обробка інформації, яку несе подія;
- зміна певного атрибута об'єкта;
- обчислення;
- генерація події для деякого екземпляра деякого класу (можливо, для самого себе);
- генерація події, що має передаватися зовнішнім щодо даного домену об'єктам, як-от людина-оператор, інша система, фізичний прилад тощо;
- прийом повідомлення про події від зовнішніх об'єктів;
- взаємодія з двома специфічними об'єктами — таймером та системним годинником.

*Таймер* — це механізм вимірювання інтервалу часу, який вважається вбудованим у даний метод системним об'єктом, що не потребує визначення.

Атрибутами цього об'єкта є:

- унікальний ідентифікатор екземпляра таймера;
- залишок часу (інтервал часу, через який буде подано сигнал про настання певної події);
- мітка події, яка настане, коли залишок часу буде дорівнювати нулю;
- ідентифікатор екземпляра об'єкта, для якого встановлюється таймер. Екземпляр таймера встановлюється для окремого екземпляра певного керованого об'єкта (наприклад, бак накопичувача, духовка

шафа печі, шахматист Іванчук) для повідомлення про настання події, даними якої є значення атрибутів таймера. Окремі події передбачено для скидання таймера в нуль та знищення таймера.

Приклад моделі станів з використанням таймера див. на рис. 3.4.

Альтернативною графічній нотації ДПС є таблична нотація — так звана *таблиця переходів у стани* (ТПС). Кожний з можливих для класу об'єктів станів представляється рядком ТПС, а кожна з можливих подій — стовпчиком. Клітина ТПС визначає стан, в який переходить об'єкт, якщо відповідна стовпчику подія відбудеться, коли він перебував у стані, котрий відповідає стовпчику. При цьому можлива ситуація, коли певна комбінація події — стан не веде до зміни стану екземпляра об'єкта або неможлива. Тоді у клітині ТПС відповідно зазначається: “подія ігнорується” або “не може бути”, або клітина залишається пустою.

У табл. 3.1 наведено приклад ТПС, що відповідає ДПС, зображеній на рис. 3.4.

Таблиця 3.1 Приклад таблиці переходів

	П1	П2	П3	П4	П5	П6	П7	П8
С1	2							
С2		3						
С3			4					
С4				5				
С5					6			
С6						7	2	
С7								1

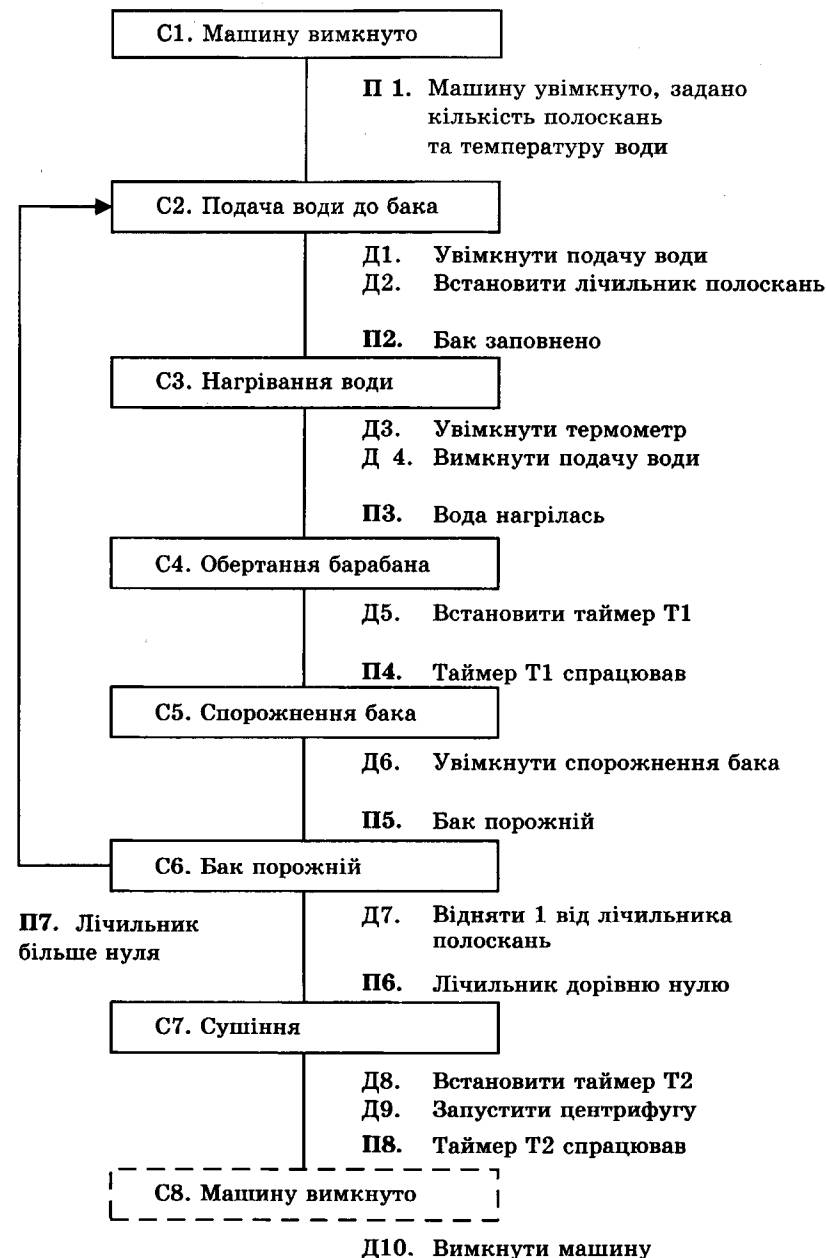


Рис. 3.4. ДПС автоматичної пральної машини

Якщо вибирати між ДПС і ТПС, то аргументом на користь першої нотації — ДПС — є її наочність та визначення дій, тоді як друга з них — ТПС — дозволяє зафіксувати всі можливі комбінації станів — подій і забезпечити повноту та несуперечність представлення вимог.

Системний годинник — це також вбудований у метод об'єкт, атрибути якого — показники системного часу (години, хвилини, день, місяць, рік) — можна читати.

Все, що було сказано вище, стосується окремих об'єктів як базових складових — компонентів архітектури системи в цілому.

Важливим принципом об'єднання компонент у систему є наявність для компонент спільних подій, при чому найчастіше одна з компонент породжує подію, а інші на неї реагують. На цьому принципі базується спосіб об'єднання окремих об'єктів у систему.

Програмна система в цілому розглядається як взаємодія об'єктів, що моделюється як обмін між об'єктами системи та зовнішніми об'єктами-повідомленнями про настання певних подій та даних до них. При цьому зовнішні події, про які система не робила запиту, вважаються такими, що запускають систему. Зовнішні події, чекання на які передбачено в системі, представлені як події, прохання повідомити про які надсилаються системою до зовнішніх об'єктів (як запит), котрі у відповідь, у свою чергу, надсилають повідомлення про настання подій до об'єктів системи.

Оскільки поведінка об'єкта представлена відповідною ДПС, то поведінка системи в цілому представляється як схема взаємодії окремих ДПС. Кожній з них присвоюється назва і вони зображаються на схемі овалом з цією назвою. Овали пов'язані між собою стрілками, що відповідають повідомленням про події, які пов'язують окремі ДПС. На стрілці вказується мітка події, а напрямком стрілки відповідає напрямку передачі повідомлення. Зовнішні об'єкти позначаються прямокутними рамками з їхніми назвами.

Приклад взаємодії моделей поведінки об'єктів наведено на рис. 3.5.

### 3.4.3. Модель процесів

Модель станів об'єктів, за допомогою якої представляються вимоги до поведінки системи (див. п. 3.4.2), передбачає у своєму складі опис певних дій, котрі супроводжують зміни станів об'єктів. Дії є алгоритмами, що виконуються системою як реакції на події і визначають її функціональність. Розуміння вимог до системи передбачає і розуміння зазначених вище дій, інколи досить складних. Способом, що пропонується для подолання труднощів розуміння дій у даному методі, є декомпозиція їх на окремі складові, які отримали назву процесів. Послідовність виконуваних процесів утворює потік керування; водночас процеси під час виконання обмінюються даними, що утворюють потоки даних; два зазначені типи потоків пропонується вико-

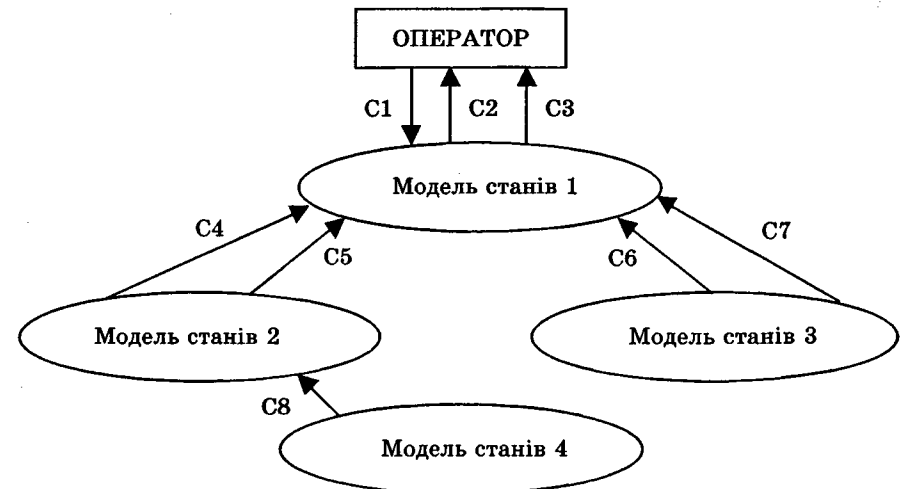


Рис. 3.5. Схема взаємодії моделей поведінки об'єктів

ристовувати як моделі алгоритмів дій системи, для представлення котрих у даному методі передбачено спеціальну нотацію, якій присвоєно назву діаграми потоків даних дій (ДПДД).

Як джерела даних допускаються:

- атрибути об'єктів (що звичайно зберігаються в архівах — файлах або базах даних, які існують і після завершення роботи системи);
- системний годинник як показник системного часу;
- таймери (див. п. 3.4.2);
- дані подій;
- повідомлення зовнішніх об'єктів (людей-операторів, приладів тощо).

Правила побудови ДПДД подано нижче:

- кожному з ДПС станів може відповідати тільки одна ДПДД;
- процес зображається на ДПДД як овал, всередині якого подано зміст або назву процесу;
- потоки даних зображено як стрілки, на яких вказуються ідентифікатори даних, що передаються від процесу до процесу; напрямком стрілки до овалу позначає дані, які є входами до процесу, напрямком від овалу — виходи;
- джерела даних зображено як прямокутні рамки чи рамки з відкритими сторонами;
- якщо джерелами даних є архівні об'єкти, відповідні потоки маркуються назвами атрибутів об'єктів, що передаються потоками, при цьому назва відповідного об'єкта може не вказуватися;
- потоки даних від таймера маркуються назвою таймера;

— потоки даних від системного годинника маркуються назвами показників часу (час, година, хвилина, день тощо);

— подія, повідомлення про яку отримує процес, зображається як стрілка, котра маркується назвами даних подій;

— якщо процес, який створив подію, та процес, який приймає повідомлення про подію, обидва належать до тієї самої ДПДД, відповідний потік пов'язує такі процеси;

— якщо подія, яку створив процес певної ДПДД, передається до процесу з іншої ДПДД, для першого із вказаних процесів вона позначається стрілкою, котра веде від процесу до “нікуди”, а для другого — до процесу з “невідкуди”, причому обидва рази стрілка маркується даними події, які передаються.

Процеси розрізняються за такими типами:

- так званий аксесор, що здійснює доступ до архівів;
- генератор подій;
- перетворювач даних (обчислення);
- перевірка умов.

Потоки керування на ДПДД позначаються пунктирними стрілками.

Якщо процес являє собою перевірку певної умови, при виконанні котрої здійснюється передача керування до іншого процесу, то відповідний потік керування зображається перекресленою пунктирною стрілкою.

Приклад ДПДД наведено на рис. 3.6.

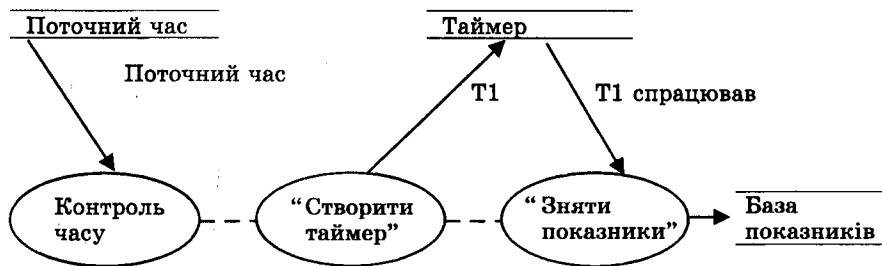


Рис. 3.6. Приклад ДПДД

До ДПДД додається неформальний опис функцій процесів, які входять до її складу. Нотація для опису подробиць дії процесів у даному методі не регламентується і залежить від смаків авторів.

Після побудови всіх ДПДД для всіх об'єктів системи доцільно побудувати загальну таблицю процесів для станів, до якої входять такі колонки:

- ідентифікатор процесу;
- тип процесу (див. вище);
- назва процесу;

— назва стану, в якому визначено процес;

— назва дії стану.

Створення такої таблиці має декілька цілей. По-перше, таблиця дає можливість перевірити несуперечність назв та ідентифікаторів процесів, по-друге, перевірити повноту визначених подій та процесів, по-третє, перевірити, чи всі визначені події генеруються певним процесом і чи всі згенеровані події обробляються певним процесом. Крім того, наявність такої таблиці дає можливість виявити процеси, спільні для кількох дій чи станів і уніфікувати їх.

Рекомендовано мати три зразки такої таблиці, кожен з яких має бути впорядковано за окремим критерієм (ключем): за ідентифікатором процесу, за моделлю станів, в яких процес використано, та за типом процесу.

#### 3.4.4. Продукти інженерії вимог за методом С. Шлеєр та С. Меллора

За даним методом, результатом проведеного аналізу вимог до створюваної програмної системи є такі продукти:

- 1) інформаційна модель системи (онтологія) у формі:
  - а) діаграми сутність — зв'язок;
  - б) опису об'єктів та їхніх атрибутів (подається неформально);
  - в) опису зв'язків між об'єктами (подається неформально);
- 2) модель поведінки об'єктів системи у формі:
  - а) діаграми переходів у стани (ДПС) або таблиці переходів у стани (ТПС);
  - б) опису дій ДПС (подається неформально);
  - в) опису подій ДПС (подається неформально);
- 3) модель процесів для станів об'єктів у формі:
  - а) діаграми потоків даних дій (ДПДД);
  - б) таблиці процесів станів;
  - в) опису процесів (подається неформально).

Сукупність перелічених продуктів вважається достатньою для переходу до процесу проектування системи, мета і методи якого подаються в главі 4.

### 3.5. Метод інженерії вимог І. Джекобсона

#### 3.5.1. Концепція моделі сценаріїв для збирання вимог

Метод, про який буде йти мова далі, є, на наш погляд, єдиним методом, котрий вказує послідовний підхід до виявлення об'єктів, суттєвих для домену проблемної галузі. Справді, всі методи декларують — як перший крок — виявлення об'єктів і попереджають, що вдалий склад об'єктів зумовить зрозумілість і точність вимог, але тільки цей підхід дає рекомендації щодо того, із чого починати шлях до розуміння проблеми та пошуку суттєвих для неї об'єктів.

Автори позначили свій метод як базований на варіантах (на прикладах або сценаріях) використання системи, яку має бути побудовано. Домовимось, що в подальшому ми будемо застосовувати термін *сценарій* для позначення прикладу чи варіанта використання системи.

Складність проблеми зазвичай переборюється шляхом поділу її на окремі компоненти з меншою складністю. Велика система може бути надто складною, щоб її компонентами були одразу програмні модулі, тому розробка системи за цим методом починається з осмислення її мети — тобто, для кого і для чого створюється система. Складність загальної мети виражається через окремі *складові мети*.

Складові мети можуть відповідати функціональним або нефункціональним вимогам, проектним рішенням та аргументам “за” чи “проти” інших складових мети. Вони є джерелом вимог до систем та засобом оцінки їх, засобом виявлення суперечностей між вимогами і встановлення залежностей. Складові мети може бути кваліфіковано як жорсткі (обов’язкові) та м’які (бажані), як функціональні чи нефункціональні, як погляд користувача чи замовника або як “кут зору” середовища функціонування системи. Складові мети також можуть перебувати між собою у певних відношеннях, як-от: конфліктувати, кооперуватися, залежати чи не залежати одна від одної.

Наступним кроком є визначення носіїв інтересів, яким відповідає кожна зі складових мети, та можливих прикладів задоволення складових мети як сценаріїв роботи системи, котрі представляють уявлення користувачів про призначення й функції системи, що можна вважати першою ітерацією вимог до розробки.

Отже відбувається послідовна декомпозиція складності проблеми:

- складна проблема трансформується в сукупність складових мети;
- кожна із складових мети трансформується в сукупність можливих прикладів використання системи, тобто прикладів реалізації складових мети, що позначаються як сценарії;

- сценарії трансформуються в процесі аналізу їх у сукупність взаємодіючих об’єктів. Визначений таким чином ланцюг трансформацій (проблема — складові мети — сценарії — об’єкти) відображає ступені концептуалізації, тобто досягнення розуміння проблеми, послідовного зниження складності її частин та підвищення рівня формалізації їхніх моделей.

Зазначимо, що наведені вище трансформації зазвичай відображаються в термінах базових понять проблемної області, тож онтологія домену, якщо її вже створено, активно використовується для представлення акторів та сценаріїв або твориться в процесі побудови такого представлення. Тобто, онтологія домену є складовою моделі вимог і за методом І. Джекобсона.

Вважається, що кожен сценарій запускає в роботу певний користувач, який є носієм певної мети. Абстракція особи користувача як

певної ролі — ініціатора запуску певної роботи, представленої сценарієм, та обміну інформацією із системою — називається *актором* (від англ. act — діяти). Це абстрактне поняття, що узагальнює поняття діючої особи системи (як основної, для обслуговування якої систему замовлено, так і вторинної, для службового персоналу системи). Фіксація акторів є також певним кроком визначення складових мети системи (носіями яких є актори) та постачальників завдань, для вирішення яких створюється система.

Актор вважається зовнішнім фактором системи, дії котрого мають недетермінований характер. Таким чином, підкреслюється різниця між користувачем, як конкретною особою, й актором — роллю, яку будь-яка особа може грати в системі.

В ролі актора може виступати й інша програмна система, якщо вона ініціює виконання певних робіт даної системи, тобто актор є абстракцією зовнішнього об’єкта, екземпляр якого може бути людиною або зовнішньою системою.

Актор в моделі представлений класом, а користувач — екземпляром класу. Одна особа може бути екземпляром декількох акторів (наприклад, водій та касир), але ми розглядаємо у вимогах тільки ролі та сценарії, в яких вони беруть участь.

Особа-в-ролі, або екземпляр актора — запускає ряд операцій у системі (транзакцію), які ми називаємо сценарієм. Коли користувач, як екземпляр актора, вводить стимул, стартує екземпляр цього сценарію, що приводить до виконання ряду дій відповідної транзакції, які закінчуються тоді, коли екземпляр сценарію знову чекає на вхідний стимул від екземпляра актора.

Екземпляр сценарію існує, поки він виконується. Його можна вважати екземпляром класу, описом якого є опис транзакції.

Для актора визначено такі правила:

- кожен сценарій може запускати тільки один актор;
- кожен актор може запускати кілька сценаріїв;
- взаємодія акторів в інтересах системи (тобто, як складова її функціональності) дозволяється тільки через передбачені для цього сценарії;
- актор не визначає сценарію, він лише ініціює ланцюжок подій, який визначить сценарій;
- для кожного актора визначаються (неформально) його інтерфейси з тими сценаріями, які він запускає.

Сценарій — це повне протікання подій у системі й, очевидно, має стан та поведінку. Кожна взаємодія між актором та системою розглядається як новий сценарій. Сценарій може розглядатися як об’єкт. Якщо багато запусків сценарію системи мають подібну поведінку, можемо розглядати їх як клас сценаріїв. Виклик сценарію є породженням екземпляра класу. Отже, сценарії — це транзакції із внутріш-

нім станом. Для них складаються детальні описи — вони є критичними для ідентифікації дійсних об'єктів системи.

Модель системи керується сценаріями, внесення змін має здійснюватися шляхом заміни потрібних акторів та сценаріїв, які вони запускають. Така модель відображає побажання користувачів і легко змінюється за їхньою волею. Користувач добре розуміє її, і до початку проектування на ній можна відпрацювати його проблеми.

Зокрема, відпрацьовується інтерфейс сценарію — за допомогою прототипу можна моделювати виконання сценарію.

Введення акторів дозволяє відповісти на запитання: для чого робиться система? Хто її головний користувач? Актори визначають зовнішнє оточення системи, а її внутрішня суть визначається сценаріями.

Сценарій — це повний ланцюжок подій, ініційованих актором, та специфікація реакції на них. Сукупність сценаріїв визначає всі можливі шляхи використання системи. Кожного актора обслуговує своя сукупність сценаріїв.

Можна виділити базову мету подій, суттєву для сценарію та його розуміння, а також альтернативні — в тому числі і при помилках користувача тощо.

Розглядаючи окремі сценарії, ми тим самим розподіляємо функціональність системи на окремі складові, над якими можна працювати паралельно.

Для моделі сценаріїв пропонується спеціальна графічна нотація, основні правила якої наведено нижче:

— актор позначається іконою людини, під якою вказується його назва;

— сценарій зображається овалом, всередині якого вказується його назва (що, зазвичай, відображає складові мети, які реалізуються сценарієм);

— ікона актора пов'язується стрілкою з кожним сценарієм, який запускає відповідний актор.

**Приклад 1.** На рис. 3.7 наведено приклад діаграми сценаріїв для клієнта банку.

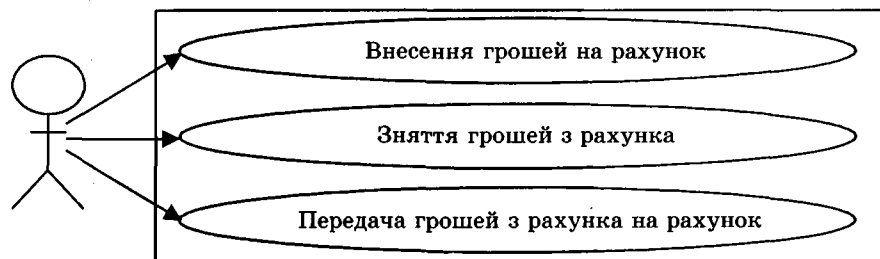


Рис. 3.7. Приклад діаграми сценаріїв для клієнта банку

Те, що клієнта визначено як актора, який запускає певний сценарій, означає, що для реалізації своєї складової мети він звертається не до касира чи клерка банку, а безпосередньо до терміналу системи (бо в іншому разі сценарії розрахунків запускав би касир, а клієнт, який взаємодіє з системою опосередковано через касира, не розглядався б як актор системи). Зверніть увагу, що всі сценарії, включені до системи, обведені рамкою, яка позначає межу системи, а актор перебуває поза рамкою, бо він розглядається як зовнішній фактор щодо системи.

**Приклад 2.** Нехай нам замовлено побудувати автоматизовану бібліотечну систему. Сформулюємо кілька складових мети її побудови:

- 1) автоматична реєстрація читача;
- 2) перевірка наявності при зверненні читача за літературою або немета та боргів з отриманої раніше літератури, термін користування якої скінчився;
- 3) фіксація книг, які замовляє читач;
- 4) фіксація факту видачі замовлень, які виконано;
- 5) фіксація повернення книжок та журналів;
- 6) організація черги відкладених замовлень, які не можна виконати через зайнятість їх іншими читачами;
- 7) повідомлення читачеві про можливість виконання відкладених раніше замовлень;
- 8) виявлення боржників і надсилання їм попереджувальних повідомлень;

На рис. 3.8 показано діаграму сценаріїв, що відповідає одному з варіантів вимог до бібліотечної системи. Розглянемо цей варіант.

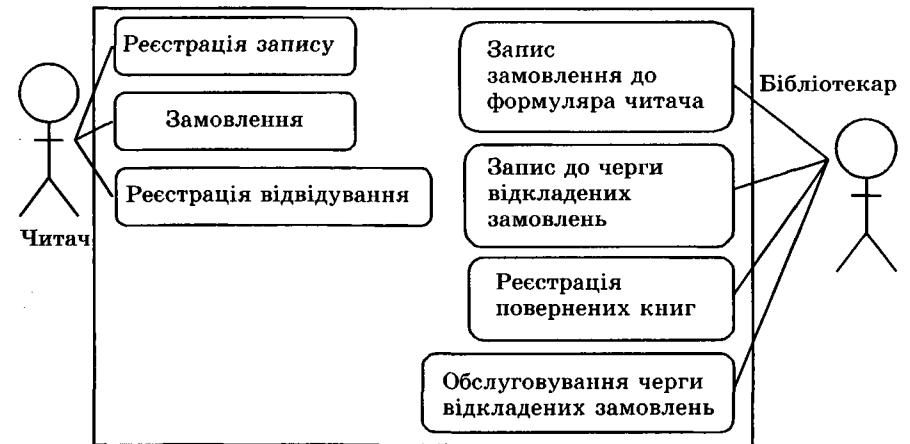


Рис. 3.8. Діаграма сценаріїв бібліотеки

Цілі 1—8 визначають ті бібліотечні процеси, які ми бажаємо автоматизувати. Вимогу до автоматизації певного процесу в системі, яку ми будемо, буде представлено одним або кількома сценаріями.

Визначимо, які актори будуть запускати сценарії.

Першим з акторів назвемо читача. Якщо ми домовилися, що саме читач запускає сценарії, котрі відповідають складовим мети 1, 2 та 3, то тим самим ми задекларували, що відвідування бібліотеки починається для нього з того, що він підходить до терміналу й запускає відповідні сценарії, інтерфейс з якими передбачає повідомлення читачем системі своїх даних під час реєстрації (складова мети 1), повідомлення абонементного номера, якщо він уже зареєстрований (складова мети 2), повідомлення свого замовлення (складова мети 3). Таке визначення треба розуміти як бажання замовника системи, щоб усе зазначене вище робилося без втручання бібліотекаря.

Другим актором обираємо бібліотекаря. Якщо ми віднесли до нього відповідальність за складові мети 4—8 (які він реалізує шляхом запуску відповідних сценаріїв), то тим самим ми висловили вимоги, щоб бібліотекар видавав замовлені примірники в руки читача, запускаючи при цьому роботу системи, результатом якої буде перенесення записів про виконані замовлення до формуляра читача, а про невиконані — до черги відкладених замовлень. При поверненні літератури запускаються сценарії відповідного занотовування у формулярі читача про звільнення поверненого примірника, який відтепер доступний, у тому числі й для відкладених замовлень.

У цьому процесі не показано діяльності з пошуку фізичних примірників замовленої літератури та переміщення їх від книгосховища до місця обслуговування читача і навпаки, бо замовник нашої системи не вимагає, щоб ця діяльність було автоматизовано, тому ми відносимо її до бібліотекаря і не відображаємо в сценаріях.

**Приклад 3.** Розглянемо систему, націлену на обслуговування роботи бібліотечної ради з формування фондів бібліотек. Передбачаються такі складові мети системи: збір пропозицій про передплату періодичних видань та придбання нових книг, погодження поданих заявок з обмеженнями фінансування, голосування про прийняття окремих пропозицій та ухвалення рішень за результатами голосування.

Першим актором нашої системи визначимо адміністратора бібліотечного фонду, а другим — члена бібліотечної ради.

Взаємодія між цими двома акторами визначається розподілом між ними відповідальності за формування фондів. Варіант такого розподілу, згідно з яким член бібліотечної ради подає пропозиції про замовлення книжок і передплатних видань і голосує за зведений список замовлень від усіх членів ради, фіксує наведена на рис. 3.9 діаграма сценаріїв. На рис. 3.9 зображено, що саме член бібліотечної ради запускає сценарії прийняття пропозицій та голосування. Це означає,

що накопичення пропозицій виконується системою автоматично. Так само і голосування виконується системою як сценарій голосування, виконання якого полягає в пред'явленні чергового замовлення членові ради, прийнятті його думки ("за" чи "проти").

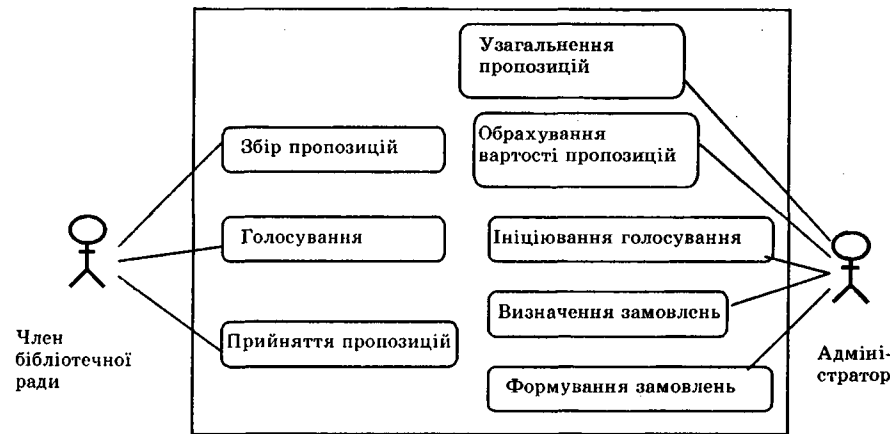


Рис. 3.9. Діаграма сценаріїв погодження замовлень

Адміністратор викликає сценарії:

- узагальнення поданих замовлень (вилучення однакових замовлень, поданих членами ради, впорядкування узагальненого списку поданих замовлень);
- обчислення сумарної вартості поданих замовлень і, якщо вона перевищує надані фінансові ресурси (бюджет), розсилання списку замовлень на голосування членам бібліотечної ради;
- обробка результатів голосування і сортування замовлень за отриманими рейтингами;
- визначення частини списку замовлень, що вкладається в наданий бюджет;
- формування бланків замовлень до видавництва та розсилання їх.

Нагадаємо, що кожен сценарій представляє певну роботу (транзакцію), яку система виконує автоматично.

**Відношення між сценаріями.** Між сценаріями можна задати певні відношення, які зображаються на діаграмі сценаріїв пунктирними стрілками, позначеними назвами відповідних відношень.

Для сценаріїв визначено два типи відношень:

1) відношення "розширює" означає той факт, що функції одного сценарію є доповненням до функцій іншого. Зазвичай воно застосовується, коли маємо кілька варіантів того самого сценарію, тоді інваріантна його частина зображається як основний сценарій, а окремі варіанти змінної частини — як розширення. При цьому основний

сценарій є стійкий щодо можливих випадків розширення його функцій — не змінюється при такому розширенні і не залежить від нього.

Наведемо типові приклади (рис. 3.10), коли доцільне застосування відношення розширення:



Рис. 3.10. Приклади розширення сценаріїв

а) для моделювання необов'язкових фрагментів сценаріїв (див. рис. 3.10, а);

б) для моделювання альтернативного перебігу подій у сценарії, що рідко відбувається (див. рис. 3.10, б);

в) для представлення ситуації, коли кілька окремих сценаріїв організовуються як акції в спеціальний сценарій типу меню (див. рис. 3.10, в).

Можемо вважати, що під час виконання сценарію, який є розширенням, переривається виконання основного сценарію (того, який розширюється), причому другий не знає, чи буде розширення і яке саме. Після завершення виконання сценарію розширення буде далі виконуватися основний сценарій.

2) відношення *“використовує”* означає, що певний сценарій може бути використано для розширення кількох інших сценаріями (аналог процедури в мовах програмування).

Обидва введені відношення є інструментом визначення успадкування, якщо сценарії вважати об'єктами. Відмінність між ними полягає в тому, що при розширенні функція розглядається як доповнення до основної і може бути зрозумілою тільки в парі з нею, тоді як у відношенні *“використовує”* додаткова функція має самостійне визначення і її можна використати будь-де.

На рис. 3.11 показано, що сценарій *“сортувати”* пов'язаний відношенням *“використовує”* з кількома сценаріями.

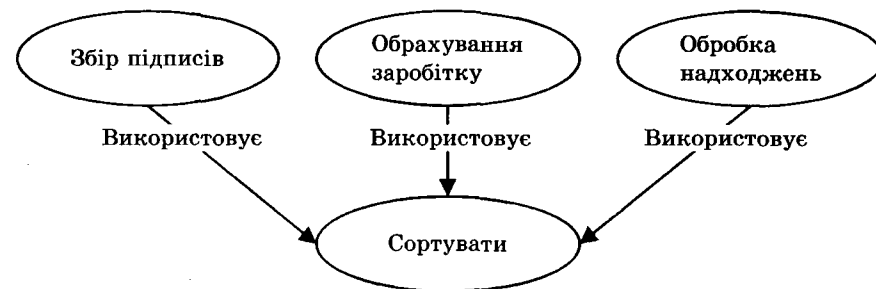


Рис. 3.11. Приклад відношення *“використовує”*

Підсумовуючи сказане вище, можна зробити висновок, що продуктом першої стадії інженерії вимог — збір вимог — є модель вимог, яка складається з трьох частин:

- 1) онтологія домену;
- 2) модель сценаріїв, яка називається діаграмою сценаріїв;
- 3) опис інтерфейсів сценаріїв.

Даний метод не регламентує жорстко нотацію для першої частини, тому можна скористатися нотацією моделі сутність — зв'язок, про яку йшла мова в п. 3.4.1.

Модель сценаріїв супроводжується неформальним описом кожного із сценаріїв, нотація якого не регламентується. Як один із варіантів, опис сценарію може бути подано як послідовність таких елементів:

- назва, яка позначає сценарій на діаграмах моделі вимог і яка є засобом посилання на сценарій;
- анотація (короткий зміст у неформальному поданні);
- актори, які можуть запускати сценарій;
- визначення всіх аспектів взаємодії системи з акторами: можливих дій актора та їхніх можливих наслідків; заборонених дій актора, якщо такі є, та їхніх можливих наслідків;

— передумови, які визначають початковий стан, тобто стан на момент запуску сценарію, необхідний для його успішного виконання, наприклад, наявність даних, яких він потребує;

— власне функції, котрі здійснюються під час виконання сценарію. Вони визначають порядок, зміст та альтернативи дій, які виконуються в сценарії, тобто його алгоритми;

— виняткові або нестандартні ситуації, що можуть з'явитися під час виконання сценарію і завадити його виконанню або потребувати спеціальних дій для усунення їх (наприклад, помилка в діях актора, яку здатна розпізнати система);

— дії, котрі є реакцією на передбачувані нестандартні ситуації;

— умови завершення сценарію;

— постумови, які визначають кінцевий стан сценарію під час його завершення.

На подальших стадіях осмислення проблеми сценарій використання трансформується в сценарій поведінки системи — до наведених елементів додаються ті, що пов'язані з конструюванням рішення цільової проблеми та нефункціональними вимогами, як наприклад:

— механізми запуску сценарію (наприклад, позиції меню);

— механізми введення даних;

— поведінка при виникненні надзвичайних ситуацій.

Отже, новий опис успадковує попередній і деталізує його.

Наступним кроком може бути сценарій тестування (див. главу 7), який успадковує попередній, але розширює його визначенням очікуваних результатів та процедури тестування.

Опис інтерфейсів також подається неформально. Слід зауважити, що саме інтерфейси визначають, якими бачать систему її користувачі, тому корисно погодити з ними вже на стадії збирання вимог усі подробиці взаємодії — як виглядає панель кожного з акторів, які вона містить елементи (меню, вікна вводу, кнопки-індикатори тощо).

Побудова прототипу системи, котра моделює реакцію цієї системи на дії актора, допоможе відпрацювати деталі й усунути взаємні непорозуміння між замовником і розробником.

### 3.5.2. Модель аналізу вимог. Визначення об'єктів

Модель вимог дає узагальнене уявлення про ті послуги (представлені сукупністю сценаріїв), які майбутня система має надавати її клієнтам (акторам). Таке представлення є предметом аналізу з метою подальшого структурування проблеми, котру розв'язуватиме згадана система. Оскільки обраною нами архітектурою є об'єктна архітектура, то результатом структурування має бути сукупність об'єктів, взаємодія яких визначає функціональність системи.

Означену сукупність знаходять шляхом послідовної декомпозиції кожного із сценаріїв на об'єкти, які відображають дії сценарію.

Декомпозиція сценарію керується намаганням провести такий вибір об'єктів, який дасть змогу забезпечити спроможність системи до адаптації в разі зміни умов або потреб використання системи. Нагадаємо, що аксіомою сучасного погляду на програмну інженерію є гасло: "Всяка працююча програмна система згодом потребує змін". Потребу в змінах готових систем усвідомлено професіоналами з програмної інженерії як об'єктивну реальність, а не лише як наслідок недоробок. Тож стратегія вибору базується на таких принципах:

- зміна вимог неминуча;

- об'єкт має модифікуватися тільки внаслідок зміни відповідних вимог до системи;

- об'єкт має бути стійким до модифікації і сприяти розумінню системи;

- стійкість до модифікацій (або стабільність) системи розуміється як їхня локальність, яка полягає в тому, що зміна кожної з вимог має вести до відповідних змін якнайменшої кількості об'єктів (в ідеалі — одного об'єкта).

Керуючись переліченими принципами, в даному методі пропонується розрізняти типи об'єктів залежно від їхньої схильності до змін. Для її оцінки простір, в якому функціонує система, структуровано такими трьома вимірами:

- інформація, яку обробляє система (її внутрішній стан);

- поведінка системи;

- презентація системи (її інтерфейси з користувачами та іншими системами).

Вибір вимірів зумовлений експертними дослідженнями динаміки змін діючих систем щодо наведених вимірів. Результати таких досліджень засвідчують:

- впродовж життєвого циклу найбільших змін зазнають вимоги до презентації системи;

- поведінка системи суттєво більш консервативна, але зазнає змін досить часто;

- характер і структура інформації, яку обробляє система, є найстабільнішим виміром щодо попередніх двох.

Тож доцільно для кожного з вимірів функціональності системи мати свою сукупність об'єктів, яка його відображає; таким поділом ми локалізуємо в тексті подання вимог найстабільніші фрагменти, наймобільніші та проміжні. Згідно із сказаним вище, ми розглядаємо три типи об'єктів:

- об'єкти-сутності;

- об'єкти інтерфейсу;

- керуючі об'єкти.

*Об'єкти-сутності* моделюють у системі довгоживучу інформацію, яка зберігається після виконання сценарію. Зазвичай їм відповідають

реальні сутності, котрі відображаються в базах даних. Більшість об'єктів-сутностей може бути виявлено з аналізу онтології проблемної області, але до уваги беруться тільки ті з них, на які посилаються в сценаріях.

Об'єкти інтерфейсу містять у собі функціональності, залежні безпосередньо від оточення системи й визначені в сценарії. За їхньою допомогою актори взаємодіють із сценаріями системи — їхнім завданням є трансляція інформації, яку вводить актор, у події, на які реагує система, та зворотна трансляція подій, які виробляє система, у повідомлення для актора. Такі об'єкти визначаються шляхом аналізу описів інтерфейсів сценаріїв моделі вимог та аналізу дій акторів із запуску кожного з відповідних йому сценаріїв. Як перше наближення, один інтерфейсний об'єкт може бути визначено для однієї пари актор — сценарій. Можна побудувати ієрархію інтерфейсних об'єктів за відношенням “містить” — наприклад, панель містить кнопки, індикатори, меню тощо.

Інтерфейси можна розподілити на два види: “з людиною” та “з системою”. При виявленні інтерфейсних об'єктів визначальним є передбачення змін — кожен пункт можливих змін у сценарії доцільно визначати як інтерфейсний об'єкт.




Керуючі об'єкти — це об'єкти, які перетворюють інформацію, введену об'єктами інтерфейсу і подану об'єктами-сутностями, в інформацію, що виводиться інтерфейсними об'єктами (іншими словами, керуючі об'єкти відповідають функціям переробки інформації). Часто вони є своєрідним “клеєм” для сполучення об'єктів, формуючи ланцюжки подій і, в такий спосіб, взаємодію об'єктів.

Такий поділ слугує складовим мети з локалізації змін у системі. При перетворенні моделі вимог на модель аналізу кожний сценарій розбивається на об'єкти зазначених вище трьох типів. При цьому один і той самий об'єкт може бути присутнім в кількох сценаріях, і важливо розпізнати такі об'єкти, щоб уніфікувати їхні функції та визначити їх як єдиний об'єкт. Критерій розпізнавання є такий: якщо в різних сценаріях посилаються на один і той самий екземпляр об'єкта, то мовиться про той самий об'єкт.

Виділяючи об'єкти, формуємо базис архітектури системи як сукупність взаємодіючих об'єктів, для кожного з яких можна дослідити зв'язок з відповідним сценарієм моделі вимог. Отже, дотримується принцип трасування вимог від моделі вимог до моделі аналізу.

Нагадаємо, що об'єкт інкапсулює в собі певні атрибути, котрі визначають стан об'єкта та його поведінку. Цю поведінку визначають операції, які об'єкт може виконувати, та стани, в яких він може перебувати.

Модель аналізу має відповідну графічну нотацію:

- об'єкти-сутності зображаються символом 
- об'єкти інтерфейсу зображаються символом 
- керуючі об'єкти зображаються символом 

Атрибути об'єктів представлені прямокутниками, поєднаними прямою лінією з символом об'єкта, при цьому на лінії вказується назва атрибута, а в прямокутнику — його тип.

Між об'єктами визначаються асоціації, які зображаються однією двоспрямованою стрілкою, на яких вказуються назви асоціацій. Серед асоціацій застосовуються найпоширеніші:

- “взаємодіє”;
- “складається з”;
- “виконує роль”;
- “успадковою”;
- “розширює”;
- “використовує”.

Зазначимо, що асоціації між об'єктами суттєво відрізняються від асоціацій у моделях даних. Другі націлені переважно на здійснення навігації в базах даних, тоді як перші означають взаємодію об'єктів.

Як приклади, на рис. 3.12 подано фрагменти моделі аналізу, що відповідають діаграмі сценаріїв бібліотечної системи, яку зображено на рис. 3.8.

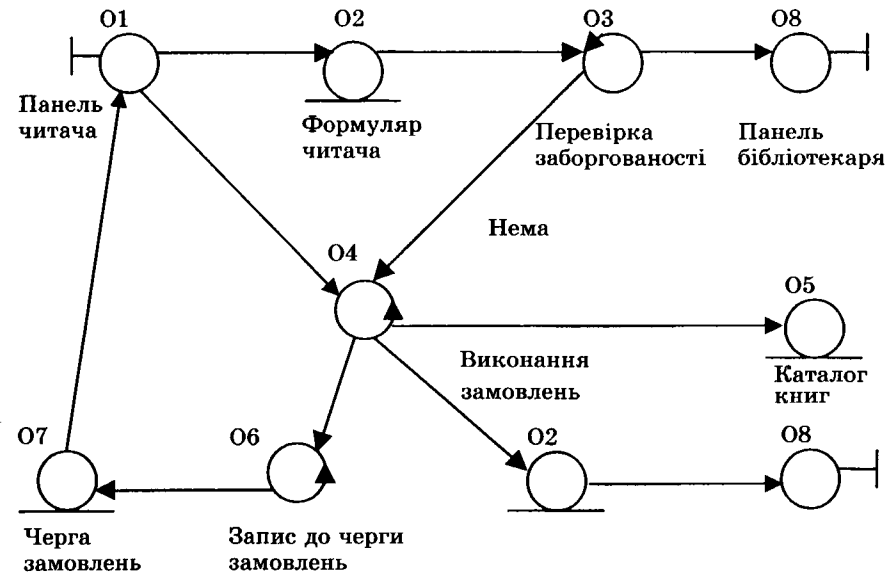


Рис. 3.12. Асоціації між об'єктами бібліотечної системи

### 3.5.3. Продукти інженерії вимог за методом І. Джекобсона

Продуктами стадії аналізу вимог за даним методом є:

- онтологія домену;
- модель сценаріїв;
- неформальний опис сценаріїв та акторів;
- опис інтерфейсів сценаріїв та акторів;
- діаграми взаємодії об'єктів сценаріїв.

Подальша деталізація проблеми за цим методом відбувається на наступних етапах життєвого циклу (див. главу 4), при цьому зберігається трасування вимог, тобто відстеження відповідності об'єктів впродовж усіх етапів розробки.

### Контрольні запитання і завдання

1. Як називається фаза життєвого циклу розробки програмного забезпечення, на якій формується контракт між замовником і виконавцем розробки?
2. Назвіть дійових осіб процесу формування вимог.
3. Назвіть джерела відомостей про вимоги.
4. Якою є послідовність кроків з урахування діючої системи в новій розробці?
5. Назвіть категорії класифікації вимог.
6. Складові мети і складові концептуального моделювання проблеми.
7. Що означає онтологія в концептуальному моделюванні проблеми?
8. Поясніть суть відношень, за допомогою яких будуються поняття: узагальнення /конкретизація, агрегація/, декомпозиція, абстракція, асоціація.
9. Які елементи моделювання динамічних властивостей доменів Ви можете назвати?
10. Назвіть елементи об'єктно-орієнтованого моделювання програмних систем.
11. У чому полягає принцип приховування інформації? Що дає цей принцип:
  - а) замовникові? б) виконавцеві?
12. Визначте операцію успадкування класів.
13. Назвіть продукти аналізу домену за методом Шлеєр і Меллора.
14. Якою є онтологія домену за методом Шлеєр і Меллора?
15. У чому суть і якою є нотація моделі станів за методом Шлеєр і Меллора?
16. У чому суть і якою є нотація моделі процесів за методом Шлеєр і Меллора?

17. У чому полягає концепція моделі сценаріїв для збирання вимог? Поняття **актор**.

18. Наведіть нотацію діаграми сценаріїв за методом Джекобсона.

19. Назвіть базові відношення сценаріїв методу Джекобсона і мету використання їх.

20. Які принципи і мета класифікації об'єктів за методом Джекобсона?

21. Якою є нотація взаємодії об'єктів у методі Джекобсона?

### Література до глави 3

1. <http://www-ksl.stanford.edu.html>
2. Шлеєр С., Меллор С. Объектно-ориентированный анализ: моделирование мира в состояниях. — К.: Диалектика, 1993. — 240 с.
3. Jacobson I., Griss M., Jonsson P. Software Reuse. — N.-Y.: Addison-Wesley, 1997. — 497 p.
4. <http://www.rational.com.uml.html>
5. Андон А.И., Яшунин А.Е., Резниченко В.А. Логические модели интеллектуальных информационных систем. — К.: Наук. думка, 1999. — 320 с.

## Глава 4. ПРОЕКТУВАННЯ ПРОГРАМНИХ СИСТЕМ

### 4.1. Проектування як процес

Проектування — це етап життєвого циклу розробки програмних систем, наступний після інженерії вимог. Завданням цього етапу є перетворення побажань замовників системи, які ми представили як моделі вимог, у проектні рішення, що забезпечать здійснення згаданих побажань у формі відповідної системи програмування. Таким чином, під час проектування виконується трансформація простору вимог у простір проектних рішень. При цьому можна виділити процеси, котрі можна вважати відносно незалежними одне від одного і виконувати як послідовно, так і паралельно, окремими командами виконавців. Це такі процеси:

- *концептуальне проектування* полягає в уточненні розуміння й узгодження деталей вимог;
- *архітектурне проектування* полягає у визначенні головних структурних особливостей системи, яку будують;
- *технічне проектування* полягає у відображенні вимог середовища функціонування і розробки системи та у визначенні всіх конструкцій як композицій компонент;
- *детальне проектування* полягає у визначенні подробиць функціонування та зв'язків для всіх компонент системи.

В основі проектування будь-якого продукту лежить парадигма подолання складності загального завдання шляхом декомпозиції цільового продукту на окремі його складові або компоненти. Це твердження діє і для програмних систем як продуктів програмної інженерії.

У попередніх главах було зазначено, що для сучасного стану розвитку програмної інженерії домінуючою є об'єктно-орієнтована парадигма, за якою будь-яка система розглядається як сукупність взаємодіючих об'єктів, тож усі наведені вище підпроцеси проектування ми будемо розглядати щодо тих об'єктів, які було визначено на попередньому етапі життєвого циклу розробки — етапі інженерії вимог.

### 4.2. Концептуальне проектування

Уточнення вимог та адекватного розуміння їх замовником провадиться в кількох напрямках. Зупинимось на головних з них, котрі відповідають вимірам простору визначення завдань системи, наведеним у п. 3.5.2.

#### 4.2.1. Уточнення даних

1) Уточнення інформації, яку система обробляє як дані, потребує відповідей на низку запитань, котрі має бути отримано як результат даного підпроцесу:

— по-перше, визначаються джерела надходження даних та те, яка сторона несе відповідальність за їхню достовірність — джерело чи система, що їх отримує. Якщо відповідальною визначено систему, необхідно доповнити проект системи відповідними блоками верифікації даних;

— по-друге, уточнюються атрибути даних;

— по-третє, визначаються способи матеріалізації зв'язків між об'єктами у формі відповідної організації даних. На цьому питанні зупинимось більш докладно.

Нагадаємо, що суттєві ознаки об'єкта представляються його атрибутами, причому один атрибут або певна сукупність атрибутів є визначальним для ідентифікації екземпляра об'єкта і називається *ідентифікатором* об'єкта. Об'єкти можуть перебувати у відношеннях або зв'язках між собою.

Коли в онтології домену визначено, що об'єкт **A** перебуває в певному відношенні чи зв'язку з об'єктом **B**, то мова йде про зв'язки між екземплярами об'єктів. Кількість екземплярів, які можуть брати участь у зв'язку з кожної сторони (**A** та **B**), визначає тип зв'язку (див. п. 3.4.1). Зазвичай зв'язок матеріалізується за допомогою додаткових атрибутів даних або навіть структур даних.

Будемо розрізняти статичні, сталі зв'язки, котрі не змінюються або змінюються досить рідко (наприклад, чоловік — дружина, батько — син, фірма — адреса) та динамічні зв'язки, які мають певні стани, що можуть змінюватися протягом робочого сеансу системи.

Статичні зв'язки реалізуються шляхом додавання спеціальних атрибутів для об'єктів, котрі беруть участь у зв'язку. Беручи до уваги, що переважною моделлю представлення даних на сьогодні є реляційна модель, в якій не дозволяється мати множинні (повторювані) значення атрибутів, згадане додавання виконується за такими правилами:

а) при зв'язку **1 : 1** додатковий атрибут може визначатися для одного будь-якого з пов'язаних об'єктів і містити ідентифікатор екземпляра, який бере участь у зв'язку. Наприклад, якщо мова йде про паркування автомобілів у боксах гаража, то ідентифікатор боксу може

вказуватися як атрибут автомобіля, при цьому ідентифікатор автомобіля не буде належати до атрибутів боксу; можливий і варіант навпаки, коли додатковий атрибут для представлення зв'язку надається боксу;

б) при зв'язку  $1 : n$  додатковий атрибут надається об'єкту,  $N$  екземплярів якого можуть брати участь у зв'язку. Наприклад, якщо працівник має кілька станків, то такий зв'язок представляється шляхом надання об'єкту "станок" додаткового атрибута "власник";

в) при зв'язку  $n : m$  утворюється додатковий об'єкт, так званий асоціативний об'єкт, який фіксує два екземпляри (по одному для кожного з об'єктів), котрі беруть участь у зв'язку. Такий об'єкт, окрім своєї назви, має першим атрибутом ідентифікатор першого з пов'язаних екземплярів об'єктів, а другим атрибутом — ідентифікатор екземпляра другого.

Приклади відображення зв'язків *a, б, в* наведено на рис. 4.1.

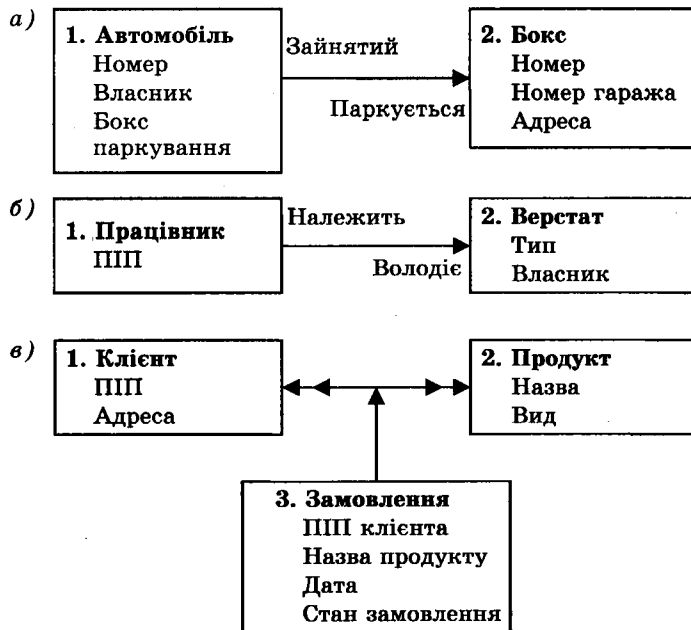


Рис. 4.1. Приклади відображення зв'язків об'єктів

Для певних завдань зв'язки між об'єктами можуть еволюціонувати із часом, і фаза еволюції може суттєво впливати на хід виконання відповідного завдання. Для таких видів зв'язку обов'язково будується асоціативний об'єкт, для якого визначається модель станів (див.

п. 3.4.2). Для представлення стану асоціативного об'єкта до складу його атрибутів додається атрибут, який фіксує його поточний стан. Нехай для прикладу на рис. 4.1, в замовлення може перебувати в станах: "отримано", "готується", "чекає розрахунку", "сплачено", "готове" тощо, тоді до атрибутів асоціативного об'єкта додається, крім зазначених на рис. 4.1, в, ще й атрибут "стан виконання замовлення", який у певні моменти часу може приймати одне з наведених вище значень. Зазначимо, що серед дій, котрі супроводжують переходи в стани для моделі станів зв'язків, мають бути операції створення нового екземпляра асоціативного об'єкта (коли нова пара екземплярів вступає у зв'язок) та його знищення (коли зв'язок переривається).

#### 4.2.2. Уточнення інтерфейсів

Узгодження інтерфейсів з потенціальними користувачами системи на ранніх стадіях життєвого циклу розробки має на меті вирішення двох завдань:

— по-перше, допомогти користувачеві перевірити його розуміння системи і отримати його схвалення складових мети та функцій цієї системи;

— по-друге, дозволити розробникові впевнитися, що запропоновані ним правила взаємодії користувача та системи задовольняють обидві сторони з погляду взаєморозуміння, ефективності і швидкості сприйняття та реагування.

Організація інтерфейсів базується на певних ключових елементах, визначення яких має передувати проектуванню конкретних екранів та форматів обміну даними. Це такі елементи:

1) метафори, поширені в домені користувача. Під цим терміном розуміємо значущі терміни, образи та поняття, які є для нього знайомими й зрозумілими або вивченими;

2) ментальна модель організації й представлення даних, функцій та ролей;

3) правила навігації (перегляду) даних, функцій та ролей;

4) візуальні прийоми демонстрації перед користувачем елементів, визначених у п. 1—3;

5) методи взаємодії, які прогнозуються як відповідні уподобанням користувачів майбутньої системи.

У визначенні наведених елементів мають братися до уваги аспекти культури, до якої належать потенціальні користувачі системи, як-от норми, традиції, звичаї та міфи домену, й притаманні його дійовим особам критерії уподобань.

Перелічені елементи використовуються при проектуванні конкретних інтерфейсів.

Окремо наголосимо на необхідності використання в меню та в іконах інтерфейсів мовних форматів і правил їхніх трансформацій,

обчислень валютних одиниць, метричних показників систем вимірювання (гривні, долари, рублі, метри, дюйми, бушелі тощо), які є звичними для кола користувачів системи, що буде створено. Вони створюють для користувачів певний психологічний комфорт.

Якщо необхідно зробити систему “полікультурною”, тобто здатною до адаптації, необхідно текстуально виділити чутливі до культурного середовища елементи, які потребують заміни: ікони, звукові повідомлення, тексти. Слід пам’ятати, що довжина текстів суттєво залежить від лаконізму обраної мови спілкування (серед них англійська, мабуть, найбільш лаконічна). Тому довжина текстових повідомлень та вікон вводу текстів має бути параметром налаштування.

Зазначимо, що правила навігації мають враховувати традиції читання (зліва направо або навпаки).

Як бачимо, питань виникає багато, тому загальною рекомендацією є побудова всіх екранних та друкованих форм системи й “програвання” з користувачем різних їхніх варіантів, щоб обрати ті, котрі відповідатимуть його уподобанням. При цьому зазвичай доводиться робити вибір між різними несумісними характеристиками інтерфейсу, як, наприклад, зручність доступу та забезпечення конфіденційності, швидкодія та складність обробки, легкість сприйняття повідомлень (об’ємна графіка, звуковий супровід тощо) та вартість розробки.

Для побудови інтерфейсів є широкий вибір методів і засобів. Більшість з них базується на фіксації певних класів об’єктів інтерфейсу ( вибір з меню, заповнення екранних форм, пряме маніпулювання — так званий стиль “зацепи та підтягни”) та на засобах монтування їх у програмну систему як інтегрованих з нею блоків або автономних підсистем.

На закінчення нагадаємо, що ми вели мову про інтерфейси об’єктів, які було визначено під час аналізу вимог і зафіксовано у відповідних моделях. Інтерфейси об’єктів означають операції, які може виконувати об’єкт, та повідомлення, які він може надсилати або отримувати.

#### 4.2.3. Уточнення функцій обробки даних

Для зафіксованих у моделях вимог об’єктів уточнюються склад і зміст властивих їм операцій (методів) і уточнюються схеми взаємодії об’єктів.

Зміст операцій, які здатні виконувати об’єкти, може бути розкрито за допомогою діаграм потоків даних для кожної з операцій (див. п. 3.4.3).

Взаємодія об’єктів організовується шляхом обміну повідомленнями, у відповідь на які об’єкти виконують відповідні операції і змінюють свій стан (див. п. 3.4.2) або посилають повідомлення іншим об’єктам.

Для уточнення поведінки об’єктів можна рекомендувати використання моделей у вигляді діаграм, котрі відображають аспекти

взаємодії об’єктів. Такі діаграми входять до складу методу UML, про який ми згадували у п. 3.3 і який детальніше обговорюється в главі 5. Зокрема, моделі поведінки об’єктів викладено в п. 5.4.

Вочевидь, усі уточнення, зроблені щодо даних, інтерфейсів та поведінки об’єктів сценарію можуть привести до необхідності перегляду моделей аналізу вимог або навіть і складу об’єктів. Важливо наголосити, що всі необхідні корекції слід починати з корекції продуктів етапу інженерії вимог — моделі вимог, моделі аналізу вимог та інших, причому витрати на пошук місць локалізації потрібних корекцій у згаданих моделях тим менші, чим повніше забезпечується трасування вимог.

#### 4.2.4. Уточнення нефункціональних вимог

Вимоги, які називають нефункціональними, відображають здебільшого певні обмеження, накладені організацією або середовищем використання системи (див. п. 3.1). Різновидів нефункціональних вимог досить багато, але, зважаючи, що вони пов’язані з багатьма застосуваннями комп’ютерних систем і для них розроблено чимало готових рішень, є сенс вивчити можливість використання цих рішень у проекті, що розробляється. Можна стверджувати, що для різновидів нефункціональних вимог завдання їхньої реалізації становлять окрему спеціальну проблемну галузь, в моделюванні якої може бути застосовано ті самі методи, котрі було запропоновано в попередніх розділах цієї книги для моделювання доменів проблемних галузей прикладних застосувань. Серед таких назвемо вимоги секретності, вимоги безпеки, відмовостійкості, корпоративну роботу над спільними ресурсами тощо.

Будуючи моделі вимог для зазначених вище доменів, слід мати на увазі, що фактично вони використовуються у багатьох прикладних застосуваннях і можуть розглядатися як незалежні від прикладних застосувань автономні аспекти розгляду систем програмування. Для них напрацьовано чимало національних, корпоративних та відомчих стандартів, які, зокрема, фіксують відповідні онтології, можливі стимули та стани тощо. Тому, починаючи моделювання цих аспектів, треба дотримуватися відповідних стандартів. Це не лише дозволить зекономити зусилля з моделювання, а й створить передумови для використання готових програмних продуктів на подальших етапах життєвого циклу розробки. Детальніше про використання готових напрацювань у створенні програмних систем див. у главі 12.

Результатом уточнення згаданих типів нефункціональних вимог має бути розширення напрацьованих на етапі інженерії вимог моделей специфічними доповненнями як відповідних об’єктів, їхніх операцій чи зв’язків.

### 4.3. Архітектурне проектування

Архітектурне проектування полягає у визначенні головних структурних особливостей системи, яку будують, а саме: складу компонент, способів їхньої композиції, обмежень на їхні сполучення.

Сучасні програмні системи — це досить складні композиції різноманітних функцій, яким відповідають програмні модулі. Водночас є тисячі готових програмних продуктів, котрі можна включити в будь-яку програмну систему для виконання чітко визначених функцій. При цьому примітивні функції можуть складати композиції, які виконують певні узагальнені функції, ті, в свою чергу, можуть пов'язуватися в нові композиції тощо. Для того, щоб сукупність готових до використання засобів можна було переглянути й зрозуміти, введено певну поширену їхню структуризацію, що полягає в такому (див. рис. 4.2).

ПРИКЛАДНІ СИСТЕМИ
Специфічні для бізнесу компоненти
Загальносистемні компоненти <i>Інтерфейс з універсальними системами програмної інженерії</i>
Системні компоненти <i>Інтерфейс з обладнанням</i>

Рис. 4.2. Пошарова архітектура напрацювань у програмній інженерії

До першого, нижчого шару відносять системні компоненти, котрі здійснюють організацію взаємодії з так званими периферійними пристроями комп'ютерів (принтери, клавіатура, сканери, маніпулятори тощо). Вони здебільшого використовуються при побудові операційних систем і не потрапляють у поле зору розробників прикладних застосувань.

До другого шару відносять так звані загальносистемні компоненти або посередники, котрі забезпечують взаємодію прикладних застосувань з універсальними сервісними системами, з такими, як операційні системи, системи баз даних та знань, системи керування мережами тощо. Компоненти цього шару використовуються в багатьох прикладних застосуваннях як складові компонент прикладних програмних систем.

До третього шару відносять специфічні для певної проблемної галузі й залежні від неї компоненти, які може бути використано як складові для спектра програмних систем, призначених для розв'язання задач означеної галузі (так званої сім'ї програмних систем).

Нарешті, до четвертого шару відносять програмні системи, побудовані для вирішення конкретних задач конкретних груп споживачів інформації, заради яких, власне, і створено компоненти всіх інших шарів.

Компоненти кожного з представлених шарів використовуються, зазвичай, тільки в своєму шарі та в наступному (вищому шарі). Для кожного шару на сьогодні визначено відповідний набір професійних знань, умінь та навичок для створення й використання його компонент, що, до певної міри, визначає відповідне розшарування професіоналів у програмній інженерії.

Ведучи мову про архітектурне проектування програмних систем, ми будемо розглядати переважно бачення програмної системи як композиції компонентів третього шару, тоді як використання компонентів другого шару є предметом розгляду технічного й детального проектування (див. нижче).

Ми отримали продукт етапу інженерії вимог як сукупність об'єктів, котрі належать до певного сценарію і взаємодія яких реалізує потрібні функції цього сценарію. Спробуємо з'ясувати, чи можна вважати об'єднання сукупностей об'єктів для всіх визначених сценаріїв системи складовими архітектури цільової системи? Інакше кажучи, чи можемо ми вважати, що такі об'єкти належать то третього шару компонент і їхня композиція є наочним представленням архітектури системи?

Відповідь на ці запитання негативна з таких міркувань: для складних систем кількість виділених об'єктів може налічувати сотні, і їхня композиція не буде мати виразного та зрозумілого представлення, навіть з урахуванням тієї обставини, що об'єкти багатьох сценаріїв можуть збігатися, тому буде потрібний додатковий аналіз для ототожнення їх.

Згадаємо основні принципи, напрацьовані як рекомендації для декомпозиції складної системи на компоненти або модулі:

- для компоненти має бути чітко визначена мета, щоб можна було перевірити, чи вона її виконує;
- для компоненти має бути чітко визначено всі її входи та виходи;
- компоненти мають утворювати ієрархію, кожний рівень якої відповідає рівню абстракції розгляду системи і дозволяє приховувати певні деталі, які буде відпрацьовано на наступних рівнях. Така покрокова деталізація прийняття рішень не стільки розподіляє вирішення складного завдання на кілька вирішень простіших завдань, скільки дозволяє відкласти детальні розв'язання проблем, щоб зосередитися на розв'язанні загальних рішень;
- робота над компонентами може вестися окремими членами команди із застосуванням кількох інструментальних засобів для кількох компонент, що суттєво впливає на ефективність роботи; але при цьому інтерфейси між компонентами мають бути прозорими й узгодже-

ними, щоб інтеграція компонент в єдину структуру була можливою і базувалася на спільному розумінні проблеми. При цьому ключова якість об'єктного підходу — інкапсуляція внутрішніх дій і приховування всіх подробиць, які не стосуються правил використання компоненти — має діяти і для підсистеми як композиції об'єктів.

Враховуючи зазначене вище, можна прийти до висновку, що отримані нами сукупності об'єктів доцільно об'єднувати в підсистеми. При цьому необхідно керуватися такими міркуваннями:

1) кожна створювана підсистема має асоціюватися з певними елементами продукту інженерії вимог (як, наприклад, актор, сценарій, об'єкт тощо);

2) доцільно необов'язкові функції або часто змінювані функції виділяти як підсистеми, при цьому бажано кожну функцію, для якої прогноуються зміни вимог, виділяти як окрему підсистему, пов'язану з одним актором (бо зміни найчастіше викликаються актором). Те саме можна рекомендувати і для функцій, використання яких у системі необов'язкове (довільне) і його може не бути залежно від обставин використання системи.

3) інтерфейс підсистеми тим більше прозорий і зрозумілий, чим менше вона має взаємозв'язків з іншими підсистемами. Бажано, щоб кожна підсистема виконувала мінімум зрозуміло визначених послуг або функцій (в ідеалі — тільки одну) та мала фіксовану множину чітко визначених параметрів інтерфейсу.

Можна перелічити типи зв'язків, характерних для об'єктів:

- зв'язок за внесенням змін, коли зміна одного об'єкта потребує перегляду другого або обидва об'єкти залежать від зміни третього (наприклад, об'єкт-сутність та об'єкт управління залежать від об'єкта-інтерфейсу);

- зв'язок за управлінням, коли керований об'єкт не може виконувати свої функції без повідомлень керуючого або один об'єкт стимулює виконання операцій другого;

- зв'язок за даними, коли дані (атрибути) одного об'єкта використовуються другим. При цьому можуть передаватися тільки значення даних або, поряд зі значеннями даних, передається метаінформація щодо організації їх, необхідна для правильної інтерпретації даних.

Відокремивши змінювані й довільні підсистеми, проведемо аналіз зв'язків та залежностей, які є між об'єктами, що залишилися, з метою утворення підсистем з тісними внутрішніми зв'язками між об'єктами та прозорими зовнішніми інтерфейсами.

Способи поєднання об'єктів у підсистему можна кваліфікувати так:

1) зернисте поєднання — в підсистему збираються об'єкти, які нічим не пов'язані між собою (відповідну підсистему створено для простого укрупнення компонент архітектури);

2) логічне поєднання — в підсистему збираються об'єкти, які є функціонально незалежними, але мають якусь спільну властивість, або для яких можна встановити певне логічне відношення (наприклад, ту саму функцію реалізовано для багатьох середовищ, як-от введення даних для дисків та портів мережі або різних типів даних, як, наприклад, цілого або комплексного);

3) поєднання за часом — у підсистему збираються незалежні об'єкти, які активізуються в спільний проміжок часу;

4) комунікативне поєднання — в підсистему збираються об'єкти, які мають спільне джерело даних;

5) процедурне поєднання — в підсистему збираються об'єкти, які послідовно передають одне одному керування;

6) функціональне поєднання — коли кожний з об'єктів, що входить у підсистему, виконує частину робіт для здійснення загальної функції, яку виконує підсистема, тобто всі об'єкти виконують спільне завдання.

Ми неодноразово підкреслювали, що розробляючи систему, слід постійно пам'ятати тезу: "Всяка зроблена система з часом потребує змін". Якщо проаналізувати наведені вище способи поєднання об'єктів у підсистеми з погляду стійкості до змін, коли кожна зміна вимоги потребує відповідної корекції мінімальної кількості архітектурних компонентів, то можна зробити висновок, що всі способи 1—5 не сприяють полегшенню модифікації вимог. Що ж до функціонального поєднання, то якщо ціль, яку реалізує таке поєднання, відповідає певним вимогам у моделі вимог, трасування вимог у моделі проекту можна вважати досягнутим.

Якщо в новостворюваній системі передбачається використання готових систем (так званих успадкованих систем), їх доцільно вважати підсистемами новостворюваної системи.

Використання готових компонент або спільних компонент для підсистем потребує спеціального розгляду, який буде подано у главі 12.

Архітектурне проектування може потребувати перегляду моделі аналізу вимог. Наприклад, якщо поведінка певного об'єкта частково використовується в кількох підсистемах, таку частку доцільно виділити в окремий об'єкт або навіть у підсистему.

Виділення підсистем для дуже великих проектів є досить складною роботою і може вестися з урахуванням дещо інших критеріїв. Наприклад, якщо розробку ведуть декілька груп різних рівнів компетентності або різних рівнів забезпеченості ресурсами, або роз'єднаних географічно, розподілення на підсистеми може вестися з пріоритетним урахуванням обставин, згаданих вище. Аналогічно при наявності розподіленого устаткування кожний логічний вузол може бути асоційовано з підсистемою.

Нотації для архітектурного проектування розглядаються в п. 5.10.

#### 4.4. Технічне проектування

Технічне проектування полягає у відображенні вимог середовища функціонування і розробки системи та визначенні всіх конструкцій як композицій компонентів. На цьому етапі відбувається прив'язка проекту до технічних особливостей платформи реалізації, СУБД, організації комунікацій, наявності фактора реального часу, виконавських вимог, таких, як швидкість реагування системи на зовнішні стимули тощо.

Об'єкти моделі аналізу вимог погоджуються з урахуванням перелічених вище особливостей, формалізуються всі стимули, які посилає чи отримує об'єкт, і всі операції, що є відповіддю на зазначені стимули.

Кожний з наведених вище аспектів прив'язки може потребувати побудови допоміжних інтерфейсних або керуючих об'єктів чи корекції існуючих. До того ж може виявитися можливість використання готових підсистем, чий устрій дещо відрізняється від підсистем, які було досі визначено на основі аналізу вимог. Тоді вносяться відповідні корективи до моделі аналізу вимог та архітектури системи.

Наступним кроком проектування може бути врахування певних властивостей, які зазвичай належать до так званих показників якості. Зупинимось на деяких з них.

**Надійність функціонування.** Надійність функціонування системи можна значно підвищити, якщо передбачити і відпрацювати виняткові ситуації під час роботи системи. Тестування системи провадиться, щоб переконатися, що реалізація системи відповідає висунутим до неї вимогам (див. п. 7). Але вимоги здебільшого обумовлюють, що має робити система, тоді як важливо також обумовити, чого вона не має робити. Одним із шляхів для цього є явна фіксація ситуацій, які унеможливають правильну роботу системи (так звані виняткові ситуації).

Причинами виникнення виняткових ситуацій можуть бути: помилки користувача при зверненні до системи чи під час підготовки даних; непередбачені збіги обставин функціонування системи (невиявлені під час тестування помилки проектування); випадкові збої обладнання тощо. При цьому система може реагувати по-різному: відмовитися виконувати певну послугу, виконати її помилково або зруйнувати якісь дані. Вочевидь, для другої і третьої з перелічених реакцій неможливо передбачити наслідки, тоді як для першої можна докласти певних зусиль, щоб відновити працездатність системи, наприклад виконати один з наведених нижче варіантів робіт:

1) відновити попередній стан системи (що передував винятковій ситуації) і спробувати застосувати іншу стратегію виконання послуги;

2) відновити попередній стан системи, внести необхідні корективи і повторити виконання послуги із старою стратегією;

3) відновити попередній стан системи, сформувати повідомлення про помилку й зупинити систему в очікуванні реакції користувача.

Ми бачимо, що забезпечення надійності системи вимагає для кожної її послуги передбачення виняткових ситуацій, аналізу їхніх причин та наслідків, побудови механізму відтворення попереднього стану (для чого необхідна певна стратегія запам'ятовування поточного стану системи) та виправлення ситуації. Для забезпечення таких дій застосовуються типові засоби:

- подвійне обчислення й порівняння результатів або їхніх контрольних сум, у тому числі виконаних на різних процесорах;
- таймери, що визначають часові інтервали фіксації поточного стану;
- додаткові перевірки коректності даних, які передають актори чи зовнішні системи або окремі компоненти однієї системи.

Усі зазначені вище дії втілюються в додаткових компонентах проекту і призводять до додаткових витрат на певні надмірні перестороги, які є ціною за надійність функціонування системи. Доцільність таких витрат визначається виключно специфікою систем. Якщо наслідки помилок незворотні, наприклад, як у систем підвищеного ризику (космічні та ядерні системи, моніторинг хворих, керування реальними об'єктами), доводиться йти на дублювання процесів і додаткові перевірки. Іноді навіть використовують паралельну роботу кількох команд або взаємну перевірку комп'ютерів, що працюють паралельно, коли один з них здійснює моніторинг іншого. Так, кажуть, що для керування системою ШАТЛ сім комп'ютерів дублюють один одного.

**Переносність системи.** Під таким терміном розуміють можливість замінювати певні використовувані сервісні системи (операційні системи, системи комунікацій у мережах, СУБД тощо) шляхом локального налаштування відповідних модулів. Зазвичай йдеться про переносність щодо конкретного типу сервісних систем, наприклад, переносність щодо СУБД, переносність щодо системи файлів тощо. Для реалізації таких властивостей визначаються об'єкти, які взаємодіють з типом сервісних систем, щодо якого декларується переносність. Кожний з визначених у такий спосіб об'єктів замінюється на такий, що взаємодіє не безпосередньо із сервісною системою, а з якимсь абстрактним об'єктом-посередником, котрий здійснює трансформацію абстрактного інтерфейсу в інтерфейс конкретної сервісної системи. Об'єкт-посередник при цьому має властивість налаштовуватися на конкретну сервісну систему.

Так, на рис. 4.3 об'єкт-посередник має призначені для нього операції створення файлу, читання з файлу, записування у файл,

модифікації записів файла, знищення файла. При цьому об'єкти прикладної системи, які звертаються до цього об'єкта за виконанням певних послуг для роботи з файлом, не опікуються подробицями організації файла в конкретному середовищі його реалізації. Такі подробиці інкапсулює в собі об'єкт-посередник, який, залежно від настройки, звертається при цьому до системи файлів MS-DOS або UNIX. Таким чином, точку можливих варіацій відносно використання тієї чи іншої системи управління файлами чітко локалізовано, що дає нам право стверджувати про забезпечення стійкості системи, що будеться, відносно зміни системи даного сервісу (управління файлами).

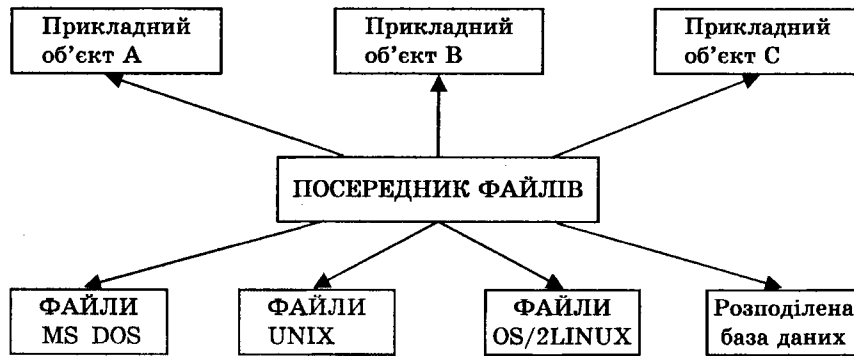


Рис. 4.3. Об'єкт-посередник

Подібні об'єкти-посередники доцільно проектувати для будь-якої передбачуваної можливості зміни вимог, яку не можна реалізувати простою заміною значень певних параметрів.

**Нотації для представлення продуктів проектування.** Продукти проектування представляються переважно в нотаціях, які базуються на моделях аналізу вимог, — більшість розглянутих нами діаграм (таких, як діаграми сутності-зв'язку, діаграми переходів у стани, діаграми потоків даних дій, діаграми класів тощо) активно використовуються як нотація для продуктів проектування. Але для них у зазначених діаграмах задіяно об'єкти проекту, що детальніше відображають не лише вимоги до розробки, а й рішення, які сприяють втіленню цих вимог. Звісно, нотації, запропоновані авторами різноманітних методів об'єктно-орієнтованого аналізу та проектування, мають свою специфіку представлення наведених вище діаграм.

Досить виразний набір діаграм для моделювання проекту має вже згадуваний метод UML, представлений у главі 5.

На рис. 4.4 подано зображення класу об'єкта *дата* за методом С. Шлеер та С. Меллора. Клас позначається прямокутником, під

верхньою лінією якого пишеться його ім'я. Шестикутниками всередині прямокутника зображено атрибути класу, кожен шестикутник поділено горизонтальною лінією, над нею зазначається ім'я атрибута, а під нею — тип, до якого він належить. Так само позначено параметри зовнішніх операцій, які може виконувати клас. Самі операції позначаються прямокутниками всередині прямокутника класу. До кожної операції зовні веде лінія, над лінією розміщуються шестикутники вхідних параметрів відповідної операції, під лінією — вихідні, а на лінії — вхідні — вихідні.

Подана на рис. 4.4 діаграма класу визначає, що клас *дата* має атрибути *день*, *місяць*, *рік* і може виконувати такі операції: створити дату, додати до дати дні, знайти інтервал (у днях) між двома датами, знайти вчорашню або завтрашню дату, перевірити, котра з двох дат передре іншої. Для всіх наведених операцій на діаграмі визначено вхідні та вихідні параметри.

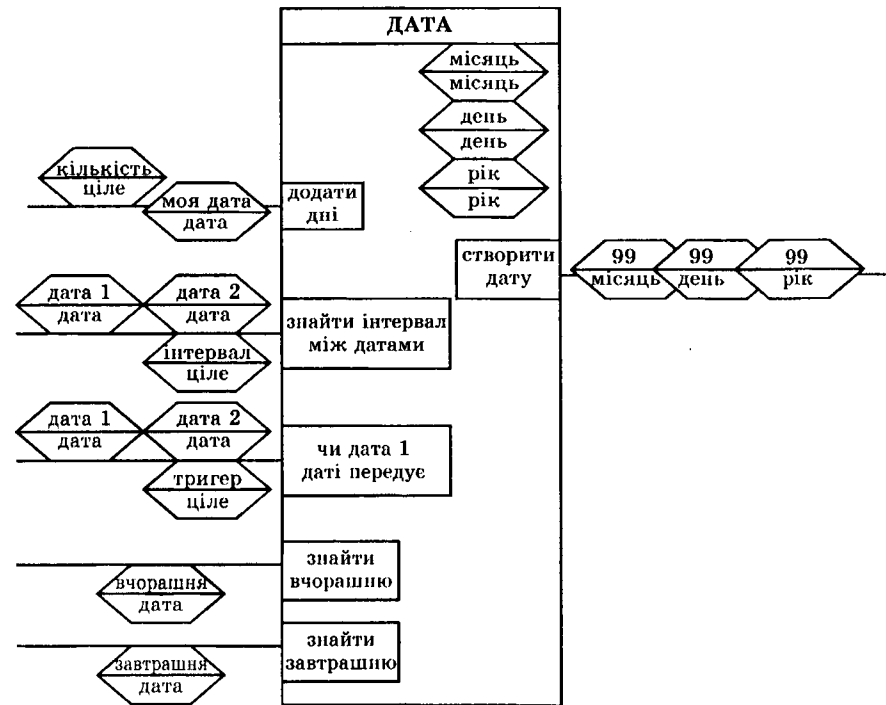


Рис. 4.4. Діаграма класу *дата*

Зазначимо, що атрибут *місяць* визначає можливі значення від 1 до 12, атрибут *день* — від 1 до кінцевої дати кожного місяця, атрибут *рік* — чотири цифри року.

Нотація проектування за методом І. Джекобсона фактично увійшла як складова до методу UML, який буде представлено в главі 5.

### Контрольні запитання і завдання

1. Визначте завдання етапу проектування програмного забезпечення.
2. Опишіть процеси етапу проектування.
3. Сформулюйте завдання концептуального проектування.
4. Які є засоби матеріалізації зв'язків у логічних структурах даних?
5. Перелічіть ключові чинники, котрі впливають на проектування інтерфейсів.
6. Назвіть нефункціональні вимоги, які треба врахувати на стадії проектування.
7. Які шари може бути виділено в сучасній архітектурі програмного забезпечення?
8. Якими аргументами треба керуватися при об'єднанні фрагментів програмного забезпечення в системи?
9. Які способи об'єднання об'єктів у системи Ви знаєте?
10. Опишіть процеси забезпечення надійності функціонування системи.
11. Які є способи забезпечення переносності системи?
12. Які нотації використовують для представлення продуктів проектування?

## Глава 5. МЕТОД UML ЯК ПОТЕНЦІЙНИЙ СТАНДАРТ ЗАСОБІВ МОДЕЛЮВАННЯ В ПРОГРАМНІЙ ІНЖЕНЕРІЇ

### 5.1. Концепція методу

Метод UML [1] (Unified Modelling Language — уніфікована мова моделювання) є рідкісним прикладом плідної кооперації групи (G. Booch, I. Jacobson, J. Rumbaugh) провідних спеціалістів з програмної інженерії і авторів відповідних методів інженерії вимог, що набули значного визнання і широко застосовуються. Дослідивши всі переваги власних пропозицій і широкий спектр конкуруючих, вони інтегрували свої зусилля, створивши новий метод моделювання, якому дали цитовану вище назву UML. UML став базовим для багатьох провідних розробників програмного забезпечення, і тепер експерти прогнозують, що він набуде статусу міжнародного стандарту як метод моделювання продуктів усіх стадій життєвого циклу розробки програмних систем.

Автори визначають свій метод як мову для специфікації, візуалізації, конструювання й документування артефактів програмних систем, а також для моделювання бізнесу.

В основу методу покладено парадигму об'єктного підходу, за якою концептуальне моделювання проблеми (дивись п. 3.2) відбувається в термінах взаємодії об'єктів:

— онтологія домену визначає склад класів об'єктів домену, їхніх атрибутів та взаємовідношень, а також послуг (операцій), які можуть виконувати об'єкти класів;

— модель поведінки визначає можливі стани об'єктів, інциденти, що ініціюють переходи з одного стану в інший, повідомлення, які об'єкти надсилають одне одному;

— модель процесів визначає дії, які виконують об'єкти.

Як і в попередньо розглянутих методах, автори UML декларують, що, враховуючи складність проблеми концептуального моделювання, її не можна розв'язати єдиною нотацією. Концептуальна модель вимог пропонується як сукупність нотацій, переважно діаграм, котрі є

візуалізацією представлення основних елементів системи в моделі. Кожна з діаграм демонструє певну підмножину інформації, яка деталізує елементи, що представляють певний аспект опису моделі та його семантику.

В комплексі сукупність включених до методу діаграм відображає найважливіші випадки функціонування системи. Перелічимо їх:

- 1) діаграми класів (Class diagram);
- 2) діаграми сценаріїв (Use-case diagram);
- 3) діаграми поведінки об'єктів, а саме:
  - а) діаграми послідовності (Sequence diagram);
  - б) діаграми співробітництва (Collaboration diagram);
  - в) діаграми активності (Activity diagram);
  - г) діаграми станів (Statecards diagram);
- 4) діаграми реалізації, а саме:
  - а) діаграми компонент;
  - б) діаграми розміщення (Deployment diagram).

Автори не закріплюють діаграми жорстко за окремими етапами життєвого циклу розробки, більшість діаграм може відображати кількома ступенями подробиць об'єкти кількох етапів: об'єкти аналізу вимог, проекту, реалізації тощо. Для цього передбачено можливість вказувати або замовчувати (залежно від стадії розробки) окремі подробиці визначення.

Кожний вид діаграм відображає різні перспективи бачення й розуміння моделі. У моделі може бути по кілька діаграм кожного з описаних видів.

Оскільки одним з головних завдань методу є досягнення взаєморозуміння між учасниками розробки, спеціальну увагу приділено коментарям. Допускається два види коментарів. Перший з них — це традиційний неформальний текст, який може бути розміщено будь-де на діаграмі в рамці з відігнутих кутами, як на рис. 5.1.

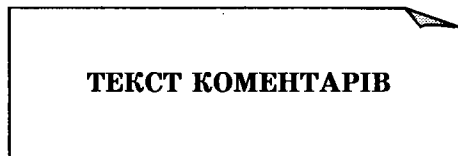


Рис. 5.1. Коментар UML

Другий тип коментаря названо стереотипом. *Стереотип* — це засіб метакласифікації елемента в UML. Він є специфічним стандартизованим коментарем щодо категорії елемента, представленого на будь-якій діаграмі, своєрідним ярликом, який характеризує зміст елемента діаграми або його призначення.

Стереотипи зображаються на діаграмах своєю назвою, яка наводиться в подвійних куткових дужках, наприклад, <<суперклас>>, <<площа>>, <<ініціалізація>>.

Певні стереотипи є фіксованими в UML і мають стандартні назви, як-от: <<актор>>, <<система>>, <<підсистема>>, <<подія>>, <<виключна ситуація>>, <<інтерфейс>>, <<метаклас>>, <<послуга>> (утиліта) та ін.

Окрім зафіксованих в UML, дозволяється визначати стереотипи за власною потребою розробника. Такі стереотипи позначають елементи моделей, типові для певного домену застосування, як, наприклад <<вимірювач>>, <<контролер>>, <<рахунок>>, <<запас>> тощо. Вони можуть вважатися інструментом розширення й адаптації UML до конкретних доменів застосувань, призначених суттєво полегшити розуміння відповідних моделей.

Зупинимось на змісті окремих діаграм детальніше.

## 5. 2. Діаграми класів

Така діаграма відображає онтологію домену і за змістом еквівалентна інформаційній моделі за методом С. Шлеєр та С. Меллора (див. п. 3.4.1): визначається склад класів об'єктів як базових абстракцій та їхні взаємовідносини. Причому нотація для опису класів забезпечує відокремлення опису функцій від опису даних, застосування принципів інкапсуляції і наслідування даних.

Діаграма має вигляд символів класів — так званих ікон та зв'язків між ними. Терміном ікона позначають стандартизоване, фіксованої форми, візуальне зображення (так би мовити, ієрогліф) певного поняття, яке легко розпізнається. Ікона класу має форму прямокутника, який може поділятися на дві або три частини. Верхня його частина обов'язкова, вона містить ім'я класу. Друга й третя частини прямокутника можуть наводитися або пропускатися і містять: друга — список атрибутів класу, третя — список операцій класу (див. рис. 5.2).

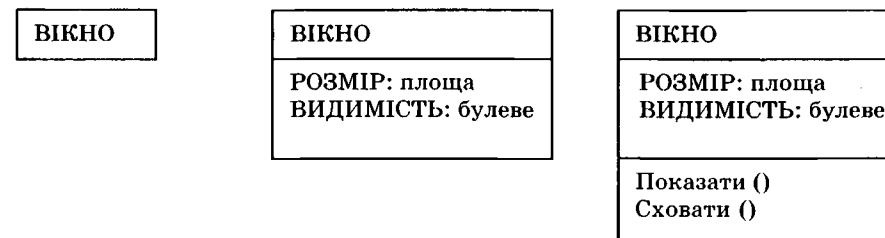


Рис. 5.2. Приклади представлення класу

Інакше кажучи, діаграма класу може відображати лише імена класів або імена та відповідні атрибути класів, або імена, атрибути та операції (методи) класів.

Атрибути можуть бути такими, типи значень яких вважаються наперед визначеними в UML, як-от: розмір, площа, кут, видимість. Останній атрибут може мати такі значення:

- спільна (public) означає, що операцію класу можна викликати з будь-якої частини програми будь-яким об'єктом системи;

- захищена (protected) означає, що операцію можна викликати тільки об'єктом того класу, в якому її визначено, або його спадкоємцями;

- приватна (private) означає, що операцію можна викликати тільки об'єктом того класу, в якому її визначено.

Воднораз користувач може визначати специфічні для нього атрибути.

Операція — це сервіс, який може надавати екземпляр класу, якщо буде відповідний виклик. Операція має назву і список аргументів.

На діаграмі може бути показано не лише класи, а й окремі їхні екземпляри. Може бути побудовано діаграму екземплярів класів. З метою відрізнити класи від їхніх екземплярів назви других у зображенні ікони класу підкреслюються. Наприклад, на рис. 5.3, а зображено клас, а на рис. 5.3, б — його екземпляр.

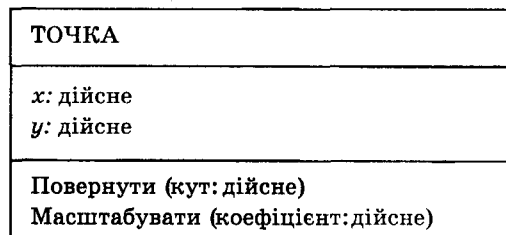


Рис. 5.3, а. Зображення класу

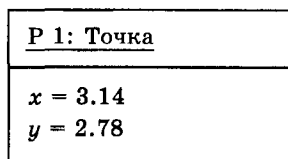


Рис. 5.3, б. Зображення екземпляра

Класи можуть перебувати у певних відношеннях або зв'язках. Розглядаються бінарні асоціації, в яких об'єкт з кожної сторони має свою роль (див. рис. 5.4).

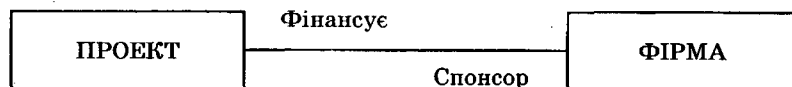
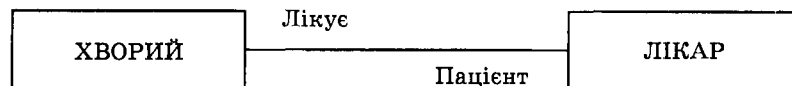


Рис. 5.4. Ролі об'єктів в асоціаціях

Для окремих типових зв'язків семантика вважається встановленою. Це такі.

**Асоціація.** Це — взаємна залежність між об'єктами різних класів, кожен з яких є рівноправним членом залежності. Для асоціації може позначатися кількість екземплярів об'єктів кожного класу, які беруть участь у зв'язку (0 — якщо жодного, 1 — якщо один, \* — якщо багато). Можуть вказуватися мінімальна й максимальна кількість, наприклад, 0,1...\* означає, що на відповідному кінці асоціації може не бути жодного екземпляра, бути один або багато. Приклад див. на рис. 5.5.



Рис. 5.5. Асоціація

**Агрегація або відношення частина-до-цілого.** Особливість цього відношення полягає в тому, що час існування об'єкта-частини збігається з часом існування об'єкта-цілого. Стрілка з ромбом на кінці, яка позначає відношення агрегації, спрямована від об'єкта-частини до об'єкта-цілого, як це подано на рис. 5.6.



Рис. 5.6. Відношення агрегації

**Наслідування підкласом властивостей суперкласу.** Може мати позначку "один до багатьох". Різновидами наслідування можуть бути відношення узагальнення й спеціалізації. Приклади див. на рис. 5.7.

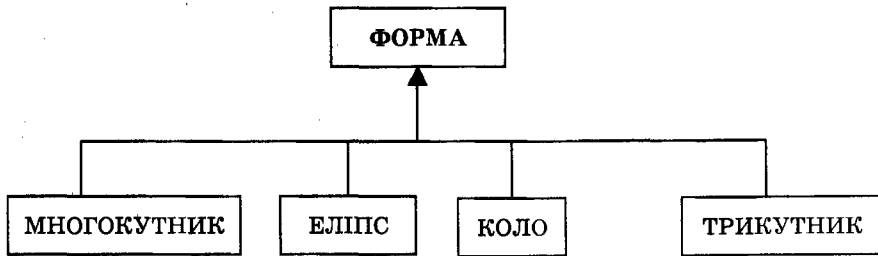


Рис. 5.7. Відношення узагальнення

**Альтернативна асоціація.** Деякий клас одночасно може перебувати у зв'язку тільки з одним елементом певної множини класів. Можливі альтернативи позначаються тим, що відповідні їм дуги перетинаються пунктирною лінією з позначкою {or} (або), як це показано на рис. 5.8.

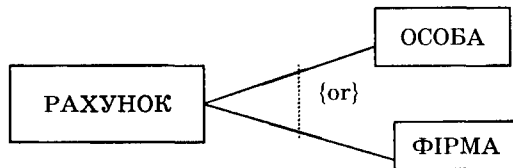


Рис. 5.8. Альтернативна асоціація

**Залежність.** Є багато видів залежностей між класами: деякий клас-клієнт може використовувати певний сервіс (операцію) іншого класу; класи можуть бути пов'язані відношенням трасування, коли один трансформується в другий унаслідок певного процесу життєвого циклу, наприклад, клас аналізу перетворюється в клас проекту, а потім у клас реалізації. Один клас може бути уточненням другого, як на рис. 5.9.

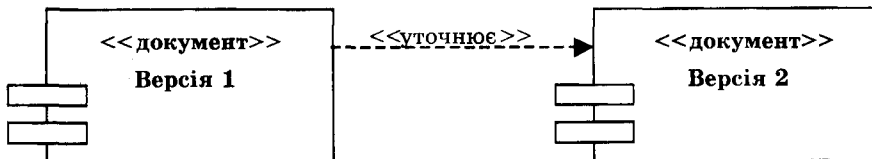


Рис. 5.9. Асоціація залежностей

**Екземплярзація.** Це залежність між параметризованим абстрактним класом-шаблоном (template) і реальним класом, який ініційовано шляхом визначення параметрів шаблону. Прикладом параметризованих класів є контейнерні класи для мови програмування C++. На діаграмі класів параметризований клас позначається так: на рамці

ікони класу зверху праворуч зображається штрихами маленький прямокутник, всередині якого подаються назви формальних параметрів шаблону, як на рис. 5.10, де *T* є параметр, що визначає тип елемента множини. На ім'я класу, який створюється внаслідок визначення параметрів шаблону, посилаються в кутових дужках після імені параметризованого класу-шаблону як префікса. Детальніше параметризовані абстрактні класи розглядаються в п. 12.3.3.

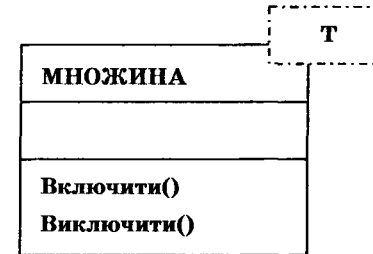


Рис. 5.10. Параметризований клас

Діаграма класів може належати до екземплярів класу, суперкласів (абстрактних класів) або підкласів (конкретних класів). У кожному з конкретних прикладів на іконі класу перед його назвою зазначається його стереотип (<<підклас>>, <<суперклас>> тощо), при цьому за умовчанням вважається клас.

Для стереотипів, які позначають відношення, фіксованими є такі: <<асоціація>>, <<наслідування>>, <<екземплярзація>>, <<узагальнення>>, <<розширення>> та інші. Крім того, користувач може вводити стереотипи, властиві специфіці його проблемної галузі, як-от: <<успадкове>>, <<контролює>>, <<є наслідком>> тощо.

### 5.3. Діаграми сценаріїв

Зміст і нотація цієї діаграми повністю збігаються з тими, що використовуються в методі І. Джекобсона (див. п. 3.5.1.).

### 5.4. Діаграми моделювання поведінки системи

Поведінка системи розглядається як обмін повідомленнями між об'єктами для досягнення певної мети. Різні сторони взаємодії об'єктів відображаються у різних діаграмах:

- *діаграми послідовності* демонструють впорядкованість взаємодії (тобто, обмін повідомленнями) в часі (уподовж лінії життя об'єктів);
- *діаграми співробітництва* демонструють ролі, які відіграють об'єкти під час взаємодії для досягнення певних цілей;

- *діаграми активності* демонструють потоки керування при взаємодії об'єктів;
- *діаграми станів* демонструють динаміку зміни станів об'єктів під впливом перебігу подій (аналогічно до діаграм переходів у стани, про які йшла мова у п. 3.4.3.).

## 5.5. Діаграми послідовності

Для наочного представлення поведінки об'єктів у сценаріях застосовується спеціальна нотація, так звана діаграма взаємодії. Для її побудови кожному з об'єктів сценарію ставиться у відповідність його лінія життя, яка відображає перебіг подій між його створенням та руйнуванням. На діаграмі вона позначається вертикальною пунктирною лінією, на верхівці якої в прямокутнику зображається назва об'єкта. Діаграма представляє всі об'єкти, які беруть участь у взаємодії. Крайня права лінія зображає зовнішнє середовище, з яким взаємодіє сценарій, тобто його зовнішній інтерфейс (зокрема це може бути інтерфейс з актором). Допускається кілька зовнішніх інтерфейсів для одного сценарію.

Порядок сусідства об'єктів не має принципового значення і обирається довільно, з намаганням забезпечити наочність взаємодії.

Вертикальний вимір діаграми відповідає осі часу, плин часу вважається вповдовж осі зверху донизу і має відносний характер: відстань на осі часу не має відповідності інтервалу реального часу, розташування вповдовж осі часу показує лише послідовність подій.

Ліворуч від осі зовнішнього інтерфейсу може наводитися перелік дій, які відбуваються в сценарії за відповідний період часу.

Ініційовані екземпляри об'єктів, поведінка яких забезпечує зазначені дії, зображаються на відповідних лініях життя подовженими прямокутниками. Довжина прямокутника відповідає інтервалу активності екземпляра об'єкта (тобто часу виконання його операцій). Отже, діаграма послідовності відображає потоки керування.

Якщо екземпляр об'єкта існував до старту діаграми, перша стрілка, яка веде до об'єкта, проводиться нижче верхівки його лінії життя.

Якщо об'єкт створюється в певний момент часу, стрілка, яка відповідає повідомленню про його створення, спрямовується до верхівки його лінії життя.

Якщо екземпляр об'єкта руйнується, нижня межа його лінії життя позначається перехрестям як символом руйнування.

Лінії життя можуть розгалужуватися, що демонструє умовні варіації поведінки об'єкта, або зливатися.

Приклад діаграми послідовності наведено на рис. 5.11.

Як уже було сказано, взаємодія об'єктів контролюється подіями, які відбуваються в сценарії і які стимулюють об'єкти для посилання

одне одному повідомлень. Повідомлення позначаються на діаграмі стрілками, над якими можуть вказуватися зміст повідомлення, параметри, які воно передає та порядковий номер повідомлення в сценарії.

Зазначимо, що вказані на діаграмі повідомлення мають узгоджуватися з видимими операціями відповідних класів.

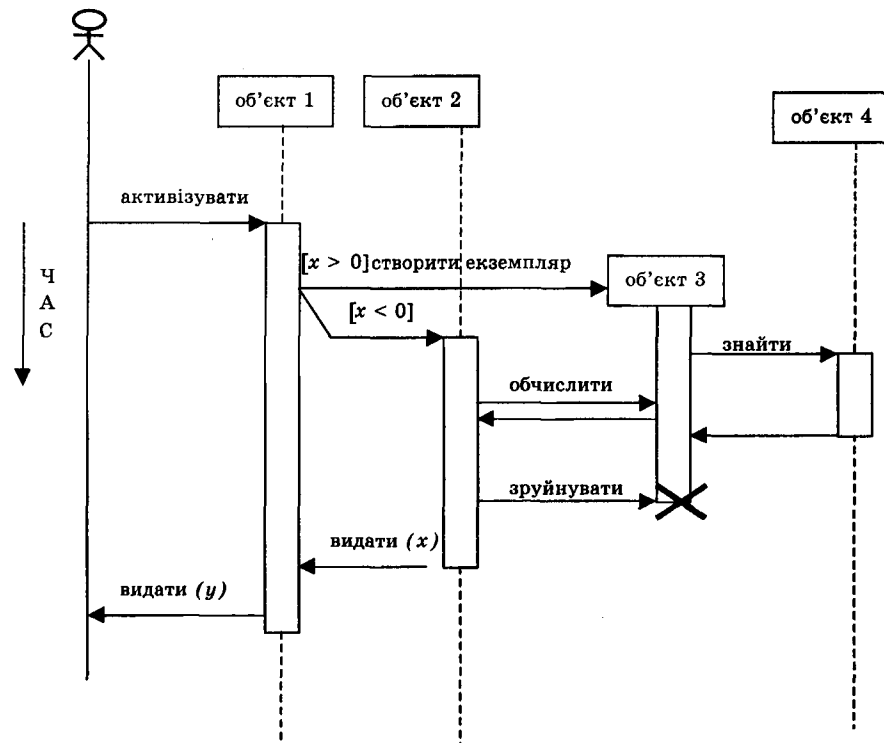


Рис. 5.11. Приклад діаграми послідовності

У наведеному прикладі екземпляри об'єктів 1, 2, 4 були до запуску сценарію і залишилися після його закінчення. Екземпляр об'єкта 3 було створено в сценарії і після виконання певних операцій зруйновано.

## 5.6. Діаграми співробітництва

Діаграми співробітництва представляють сукупність об'єктів, поведінка яких значуща для досягнення складових мети системи, та взаємовідношення тих ролей, які об'єкти відіграють у співробітництві. На даному виді діаграм моделюється статична взаємодія об'єктів,

при цьому фактор часу не враховується і не відображається на діаграмі співробітництва.

Діаграма співробітництва може бути параметризованою. Тоді вона представляє абстрактну схему співробітництва — так званий патерн, для якого шляхом довизначення параметрів можна створити певну множину конкретних схем співробітництва. Докладніше патерни обговорюються в главі 12 (див. п. 12.3.7.).

## 5.7. Діаграми діяльності

Модель *діяльності* в UML представляє поведінку системи як певні роботи, котрі можуть виконувати як система, так і актор, причому послідовність робіт може залежати від прийняття певних рішень залежно від умов, що склалися. Окрема діяльність (робота) зображається на діаграмі прямокутником із закругленими кутами. Потoki керування між роботами показуються стрілками. Якщо мова йде про прийняття рішення, то з відповідного прямокутника виходять дві стрілки, на кожній може позначатися текст умови, якій вона відповідає. Діаграма діяльності нагадує відомі блок-схеми алгоритмів та програм, зокрема передбачено відображення можливості виконувати паралельно кілька діяльностей і точки синхронізації завершення їх.

**Приклад.** Нехай пошук замовленої книги в бібліотеці має супроводжуватися кількома супутніми процедурами, а саме збором статистики про виконані та невиконані замовлення. Другі при цьому поділяються на невиконані через те, що в бібліотеці немає книг та через те, що вони в інших читачів. На рис. 5.12. представлено відповідну діаграму діяльностей. Горизонтальні лінії позначають розпаралелення та синхронізацію окремих робіт.

Діаграма може відображати той факт, що певна діяльність виконується для кожного існуючого екземпляра об'єкта, наприклад, для кожного рядка замовлення (якщо, наприклад, замовляється кілька книжок одразу). Тоді стрілка, яка веде до такої діяльності, позначається зірочкою, як на рис. 5.13.

## 5.8. Діаграми станів

Як і діаграма переходів у стани (див. п. 3.4.2.), модель *станів* UML базується на використанні розширеної моделі скінченного автомата.

Нею визначаються:

- умови переходів (застороги — guards on transitions)
- переходи, зумовлені певними подіями;
- дії при переході;
- дії при вході в стан;

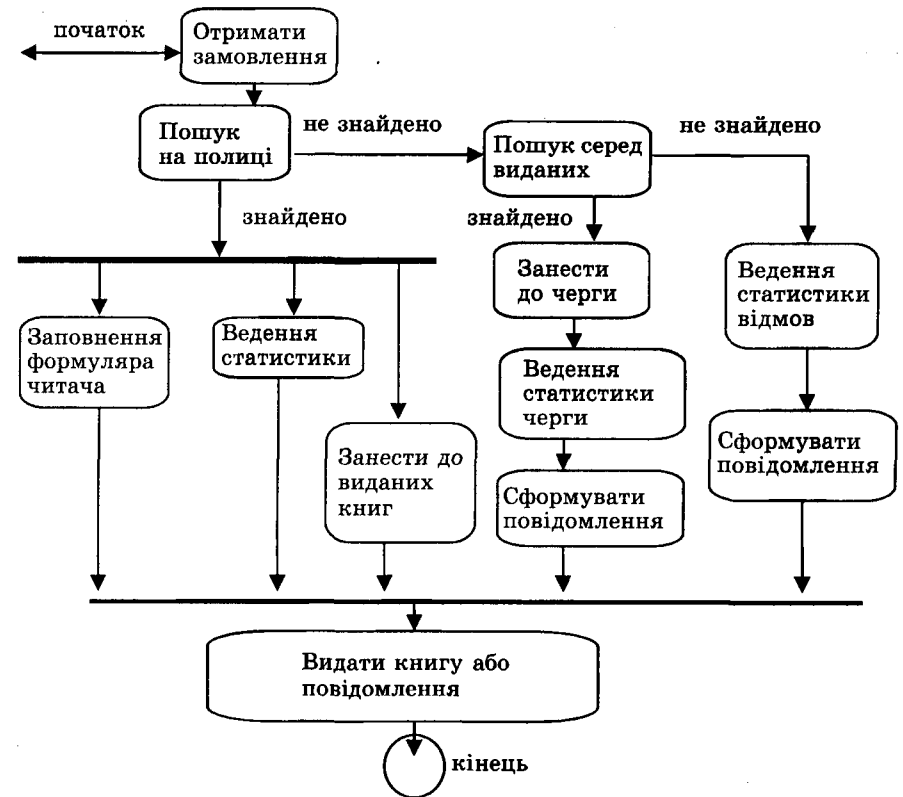


Рис. 5.12. Діаграма діяльностей сценарію замовлення книг у бібліотеці

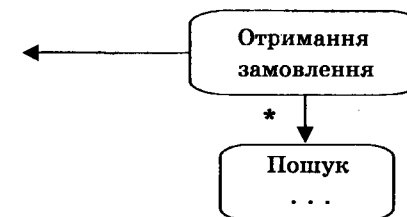


Рис. 5.13. Фрагмент діаграми діяльності для кожної замовленої книги

- діяльність, яка триває доти, доки стан є активним;
- дії при виході зі стану;
- вкладені стани;
- паралельно діючі стани.

Можуть виділятися суперстани та підстани. Вони разом із вкладеними станами дозволяють конструювати ієрархію станів системи. У даному разі подія — це назва переходу. Події можуть мати аргументи, які визначають список даних, що передаються з переходом (такі, як помилка коду або моніторинг значення). Засторога визначає умови, які потрібно виконати для здійснення переходу. Нарешті список операцій визначає деякий список функцій разом з їхніми аргументами, котрі буде викликано як результат цього переходу.

## 5.9. Діаграми реалізації

### 5.9.1. Діаграми компонент

Призначенням *діаграми компонент* є відображення структури системи як композиції компонент і зв'язків між ними так, як їх уявляє собі програміст. Це граф, вузлами якого є компоненти, а дуги відображають відношення залежності. Серед видів компонент особливого обговорення заслуговує пакет (див. п. 5.10).

### 5.9.2. Діаграми розміщення

Призначенням даної діаграми є визначення складу фізичних ресурсів системи (що позначаються як вузли системи) та відношень між ними. Системи реального часу в багатьох випадках базуються на різноманітних платформах замовника. Інженер повинен розробити не лише програмну частину, а й визначити необхідні апаратні пристрої. Ці пристрої мають органічно взаємодіяти з програмними компонентами. Ікона програмної компоненти зображається як прямокутник з двома невеличкими прямокутниками, вузол обладнання — як прямокутник зі спеціальною рамкою. На рис. 5.14 показано розміщення компонент на вузлах системи.

Для діаграми зазвичай використовуються фіксовані стереотипи: <<процесор>>, <<пристрій>>, <<дисплей>>, <<пам'ять>>, <<диск>> тощо.

## 5.10. Пакети в UML

В UML передбачено загальний механізм організації деяких елементів (об'єктів, класів, підсистем тощо) в групи. Групування можливе, починаючи від системи в цілому і до підсистем різних рівнів деталізації, аж до класів. Результат групування названо *пакетом*.

Пакет визначає назву простору, який займають елементи, що входять до його складу і є засобом посилання на цей простір; це особливо важливо для великих систем, котрі налічують сотні, а інколи й тисячі елементів, і тому вимагають ієрархічного структурування.

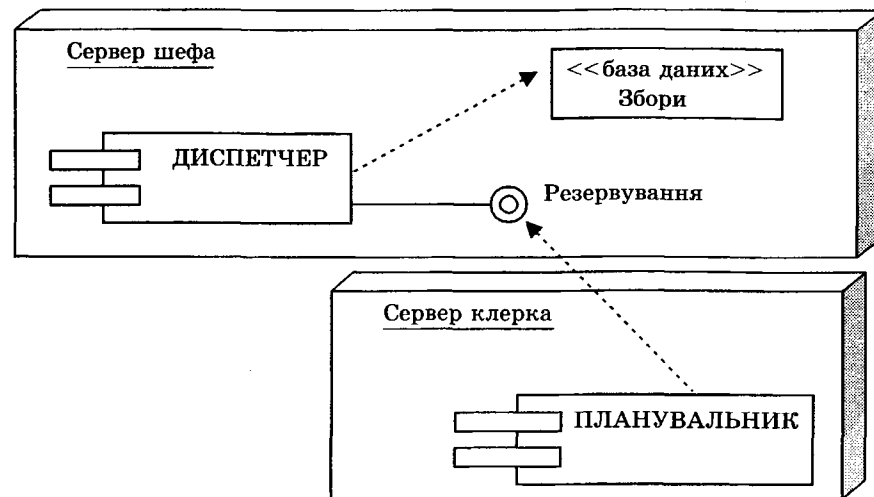


Рис. 5.14. Діаграма розміщення

*Підсистема* в UML розглядається як різновид пакета, який має самостійну функцію.

Групування в пакети може бути вкладеним, тобто складовими пакетів можуть бути класи, інші пакети та підсистеми.

Об'єднання елементів у пакети (так зване пакетування) може відбуватися з різних міркувань, наприклад, якщо вони використовуються сумісно або створені одним автором, або стосуються певного аспекту розгляду, як-от інтерфейс з користувачем, пристрої введення/виведення і под. На стадії реалізації до одного пакета може бути віднесено всі підсистеми, що в діаграмі розміщення (розглянуто вище) віднесено до одного вузла.

Призначення пакета — бути елементом конфігурації, тобто елементом, який можна включати як визначену складову композиції в побудові певної системи. На пакет можна посилатися у різних діаграмах, котрі можуть розробляти окремі команди спеціалістів. Терміном *конфігурація* будемо позначати отримання програмної системи шляхом добору окремих екземплярів модулів з визначеного наперед складу їхніх варіантів. Так, наприклад, операційна система може мати у своєму складі конфігурацію модулів, що дозволяють взаємодію з різноманітними пристроями, але лише окремі з них підключено до даного комп'ютера, для якого створюється версія операційної системи як конкретна конфігурація з визначеної множини; система керування польотом літака має у своєму складі конфігурацію модулів, що забезпечують введення показників приладів конкретного борту літака.

Пакет часто може передбачати кілька версій конфігурації його складових.

Нотація для пакета в UML подає його зображення у формі прямокутника, що містить елементи, які він включає.

Пакет, який є підсистемою або системою, зображається прямокутником з закругленими кутами.

Над прямокутником ліворуч зверху розміщується менший за розміром прямокутник, в якому подається стереотип пакета та його назва. Якщо елементи, включені до пакета, не показують, його назва розміщується у великому прямокутнику. На рис. 5.15 показано пакет, названий А 1, до котрого включено пакет В 1, клас К 1 та підсистему С 1.

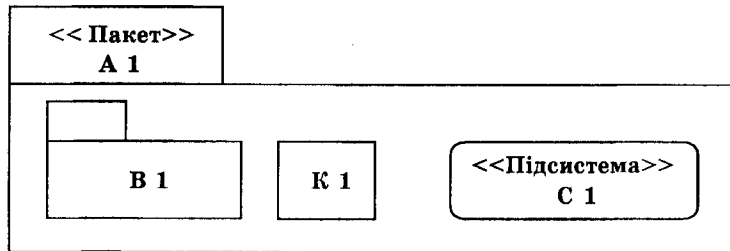


Рис. 5.15. Пакет UML

Серед фіксованих стереотипів для позначення різновидів пакета введено такі: <<система>>, <<прикладна система>>, <<підсистема>>, <<елемент конфігурації>>, <<складова системи>>, <<охоплююча система>>, <<фасад>> (facade — пакет, який є інтерфейсом іншого пакета), <<каркас>> (framework — типовий зразок взаємодії об'єктів, детальніше див. п. 12.3.8.), <<заглушка>> (stub — позначення пакета, який буде описано пізніше) тощо.

Між пакетами може бути встановлено відношення з відповідними стереотипами. Наприклад, відношення А <<імпортує>> Б означає, що визначені в пакеті Б класи можна використовувати в класах пакета А.

Нагадаємо, що дозволяється виділяти власні категорії стереотипів, в тому числі і для різновидів пакетів, котрі відповідають власним смакам чи потребам. Наприклад, стереотип <<успадковано>> може додаватися до назви пакета, який є елементом застарілої версії системи, і без переробки його включено до нової версії системи.

Пакет може належати до різних етапів життєвого циклу розробки системи — аналізу вимог, проектування, реалізації або кодування, про що може бути зазначено у відповідному стереотипі.

## Контрольні запитання і завдання

1. Які з елементів моделювання UML дозволяють визначити головні складові концептуального моделювання проблеми, а саме:
    - а) онтологію домену?
    - б) модель поведінки об'єктів?
    - в) модель процесів?
  2. Метод UML пропонує різні нотації (графічні діаграми) для різних аспектів опису проблеми. Чому не єдиною?
  3. Аналоги яких діаграм UML Вам зустрічалися:
    - а) у методі С. Шлеєр та С.Меллора?
    - б) у методі Джекобсона?
  4. Чи дозволяє діаграма класів UML відобразити відношення:
    - а) між класами об'єктів?
    - б) між екземплярами класів?
  5. Які значення може мати атрибут видимості класів, та що вони означають?
  6. Які відношення позначаються в діаграмі класів UML спеціальними графічними символами?
  7. Які відношення пов'язують сценарії системи?
  8. Які діаграми UML доцільно застосовувати для аналізу вимог?
- З якої діаграми доцільно починати?
9. Які діаграми відображають обмін повідомленнями як єдиний засіб взаємодії об'єктів?
  10. Які діаграми зручно застосовувати:
    - а) на стадії проектування?
    - б) на стадії реалізації?
- Чи можна застосовувати ті самі діаграми для кількох стадій розробки?
11. Яка роль стереотипів у нотаціях UML?
  12. Які засоби групування елементів моделювання?

## Література до глави 5

1. <http://www.rational.com.uml>
2. Фаулер М., Скотт К. UML в кратком изложении. — М.: Мир, 1999. — 200 с.
3. Буч Г., Рамбо Д., Джекобсон А. Язык UML. Руководство пользователя. — М.: ДМК, 2000. — 430 с.
4. Боггс Ч., Боггс М. UML Rational. — М: Лори, 2000. — 561 с.

## Глава 6. ТРАНСФОРМАЦІЯ ПРОЕКТУ В ПРОГРАМУ

Процес трансформації проекту в програму має кілька поширених назв: *конструювання, кодування, програмування, реалізація*. Наявність синонімів є наслідком не лише неузгодженості в термінології, а, мабуть, і тієї обставини, що кожний з термінів значною мірою відбиває певний аспект того процесу, про який ми ведемо мову в цій главі. Справді, потрібно виконати конструювання, визначивши всі складові системи як програмні модулі; потрібно запрограмувати ці модулі обраною мовою програмування, для чого слід не лише перевести нотації проекту в коди обраного засобу програмування, а й провести тестування та верифікацію отриманих кодів. А в цілому потрібно перетворити проект на працюючу програмну систему, документовану таким чином, щоб було зрозуміло, як її використовувати, інакше кажучи, реалізувати програмну систему. Для певності далі будемо вживати термін *реалізація*.

Реалізація є одним з визначальних етапів життєвого циклу розробки. Було б помилкою вважати, що всі принципи рішення розробки напрацьовано на попередніх стадіях (аналізу вимог та проектування) і далі залишилося “механістично” перекласти проект в коди працюючої системи. Якби це було так, то ринок програмних засобів був би заповнений трансляторами з проектів у коди, чого зараз не помітно. Практика засвідчує, що процес реалізації потребує високого професіоналізму і творчих зусиль програміста, наслідком яких можуть бути істотні зміни проекту.

Реалізація є першою із стадій життєвого циклу, внаслідок якої артефакти, створені людиною на попередніх стадіях, трансформуються в артефакти, які буде передано комп'ютеру, і тому мають бути для нього однозначно зрозумілими. Така трансформація потребує активного діалогу між особою, котра розуміє проект, і комп'ютером, який має трактувати відповідну до проекту реалізацію. Одна із сторін діалогу є людиною з непередбачуваною поведінкою, прихильністю до замовчування того, що вона має за “загальновідоме”, до забудькуватості та нечіткого формулювання, інша є абсолютно педантичним комп'ютером-виконавцем.

Під час перетворення проекту на працюючу систему робиться цілий ряд уточнень та формалізацій, у процесі яких програміст трансформує свої знання в зрозумілі комп'ютеру артефакти і деталізує ті можливі варіанти, які передбачено проектом. При цьому програміст має “стати на точку зору” комп'ютера, щоб зрозуміти, що він уміє і знає з того, що не обумовлено в проекті. Вміння та знання комп'ютера визначаються сукупністю інструментальних засобів, які доступні програмістові. Нині це досить широкий спектр мов програмування, генераторів програм, бібліотек об'єктних модулів, класів, функцій тощо. Тож програміст має знати все або більшість з того, що знає комп'ютер (принаймні готові напрацювання, придатні для його проблемної галузі), щоб зробити правильний вибір потрібних інструментальних засобів та готових компонент, створити ті компоненти, яких не вистачає, а також інтегрувати все в працюючу систему. Прийняття рішення про використання тих чи інших готових компонент може привести до суттєвих змін в архітектурі проекту.

З іншого боку, обрання певного інструментального засобу визначає стилістичний напрям процесу реалізації. Можна назвати три поширені стилістичні напрями:

- лінгвістичний стиль, коли програма має вигляд текстів, наближених до натуральної мови і тому звичних для більшості користувачів;
- математичний стиль, коли шляхом математичного чи логічного розмірковування можна досить точно висловити свої потреби, оскільки семантику математичних та логічних виразів точно встановлено. На жаль, цей стиль може бути застосовано лише для вузького кола добре сформульованих завдань із чітким і зрозумілим механізмом абстракції. Крім того, володіння таким стилем доступне лише відносно вузькому колу фахівців зі спеціальною освітою;
- візуальний стиль, коли для наочного представлення взаємодії об'єктів системи активно використовуються зорові образи.

Слід зазначити, що тенденції розвитку інструментальних засобів розробки програм є такими, щоб інтегрувати всі три стилі в одній розробці. Зокрема це стосується візуальних мов програмування (JAVA і под.).

Вибір і використання готових інструментів та компонент має свою специфіку в програмній інженерії, висвітленню якої присвячено главу 12.

### Контрольні запитання і завдання

1. У чому суть трансформації програмного проекту в програму.
2. Чи визначає сукупність наявних інструментальних засобів можливі напрями реалізації?

## Глава 7. ТЕСТУВАННЯ ПРОГРАМ ТА СИСТЕМ

До фундаментальної концепції проектування програмного забезпечення входять базові положення, техніки, стратегії, методи, пов'язані з етапами життєвого циклу. Вона розкриває такі основні питання: що таке тестування; відмови, дефекти та помилки; елементи тестування; процес тестування. Ця концепція пов'язана також з мовами програмування, методами проектування програм (структурне, об'єктно-орієнтоване, імперативне та ін.) і стандартами організації розробника на технологію, структуру коду тощо. Концепція базується на теоретичних, інструментальних та практичних засобах.

Теоретичні засоби визначають кодування й тестування програмного продукту. З боку кодування — це метрики коду мовою програмування (метрики Холстеда, цикломатична складність Маккейба) як формалізована теорія визначення за специфікацією програм її правильності та гарантії очікуваних властивостей кінцевого програмного забезпечення. Крім того, концепція *чиста кімната* також базується на деяких формалізмах доведення й вивчення властивостей кодування і тестування. З боку тестування — це специфікації нотацій, що використовуються при описах тестів для покриття відповідних критеріїв програм.

Інструментальні засоби — це засоби підтримки кодування й тестування (компілятори, генератори програм, засоби налагодження та ін.), а також практичні засоби планування та підготовки відповідних тестів для програм, які забезпечують обробку текстів мовою програмування.

**Тестування** — це спосіб семантичного налагодження (перевірки) програми, який полягає в опрацюванні програмою послідовності різноманітних контрольних наборів тестів з відомими результатами. Тестування ґрунтується на виконанні програми й одержанні результатів опрацювань тестів.

Тести підбираються так, щоб вони охопили якнайбільше різноманітних типів ситуацій, які виконуються при роботі елементів програми, тобто щоб програма використовувала всі допустимі шляхи свого графу передачі керування. Слабкіша вимога — виконання по

одному разу кожного розгалуження програми в кожному можливо-му напрямі.

Тестування охоплює такі основні види робіт:

- розробку програмного продукту на етапах життєвого циклу та верифікацію результатів на кожному етапі;
- упорядкування плану тестування і підготовки тестів для перевірки окремих елементів розробленої програми та програми в цілому;
- керування виконанням тестів та аналіз результатів тестування;
- повторне тестування.

**Тестування**, як правило, починається з останнього етапу проектування і включає розробку планів і тестів перевірки правильності функціонування вихідного коду. Кінцевою метою тестування є отримання сертифіката на розроблений програмний продукт. Тестування може бути виконано шляхом перевірки документів етапів аналізу та проектування компонентів.

Тестування становить від 30 % до 50 % трудомісткості робіт зі створення коду. У зв'язку з цим воно має плануватися як і інші види діяльності, починаючи з етапу аналізу і закінчуючи конструюванням.

Історично першим різновидом тестування було налагодження.

**Налагодження** означає перевірку програмного об'єкта на наявність у ньому помилок і наступне усунення їх. При цьому можуть вноситися нові помилки. Як правило, налагодження проводиться в тому разі, якщо під час розробки програмного продукту використовуються створені об'єкти-компоненти, які описуються мовами програмування і пропускаються через компілятори для перевірки переважно синтаксичної правильності.

У зв'язку з тим, що змінилися методи розробки та засоби підтримки цього процесу, на зміну налагодженню прийшли нові види діяльності із встановлення правильності створюваного вихідного коду: валідація (забезпечення відповідності продукту вимогам її замовників) та верифікація (забезпечення відповідності кодів проекту).

### 7.1. Методи тестування програм

Огляд методів тестування й верифікації програм [1—9] дозволяє вважати їх використання цілком залежним від методів проектування та стадій, з яких починається перевірка правильності функціонування результатів проектування. Як правило, проводять функціональне тестування за планами, розробленими на етапі проектування прототипу системи. Головна мета полягає в тому, щоб виявити невідповідність між фактичною поведінкою реалізованих функцій та очікуваною поведінкою відповідно до специфікації програми.

Функціональні тести проектуються за зовнішніми специфікаціями функцій, проектною інформацією етапів життєвого циклу або за спроектованим прототипом.

Тестування за зовнішніми специфікаціями провадиться за вимогами, які було сформульовано на етапі аналізу предметної галузі.

Тести при проектній інформації відображають структури даних, алгоритми й інтерфейси між окремими компонентами системи. Вони дають можливість перевірити функції компонентів і їхній взаємозв'язок.

Методи функціонального тестування поділяються на статичні й динамічні.

*Статичні методи* використовуються під час проведення інспекцій та аналізу специфікацій компонентів без їхнього виконання. До них належать методи аналізу послідовності операторів та аналізу перетворення типів даних.

*Динамічні методи* застосовуються в процесі виконання програм. Вони базуються на побудові графу, який пов'язує причини помилок з очікуваними реакціями на ці помилки. У процесі тестування накопичується інформація про помилки, яка використовується для визначення надійності програм.

#### 7.1.1. Статичні методи тестування

Техніка статичного аналізу полягає в методичному перегляді (або огляді) й аналізі структури програм, а також у доведенні правильності формальними методами.

Статичний аналіз спрямований на аналіз документів, створених на всіх етапах життєвого циклу, і полягає в інспекції вихідного коду та наскрізного контролю програми.

Інспекція полягає в спільному розгляді документів незалежними експертами за участю розробників. На початковому етапі проектування інспектування допускає перевірку повноти, цілісності, однозначності, несуперечності і сумісності документів з початковими вимогами до програмної системи. На етапі реалізації системи під інспекцією розуміють аналіз текстів програм щодо дотримання вимог стандартів і прийнятих керівних документів технології програмування.

Ефективність перевірки шляхом інспекції досягається тим, що залучені експерти намагаються глянути на проблему "збоку" і піддають її всебічному критичному аналізу. Коли проводиться наскрізний контроль, то експерти сприймають ще й словесне пояснення розв'язуваної задачі і засобів її проектування. Це та безпосередній перегляд коду дозволяють виявити помилки в логіці та в описі алгоритму.

Наскрізний контроль полягає в ручній імітації виконання програми. Розробник програми усно пояснює й обґрунтовує обрані підходи і методи реалізації, добирає тести для ручного відслідковування виконання програми.

Ці засоби дозволяють на більш ранніх етапах проектування виявити помилки або дефекти шляхом багаторазового перегляду вихідних кодів.

Методи перегляду не цілком формалізовані, вони визначаються ступенем кваліфікації експертів групи. Основні методи розглядаються нижче.

**Метод простого структурного аналізу.** Орієнтований на аналіз структури програми (потоків керування і потоків даних).

Одним із підходів до аналізу структури програми є використання теорії графів для представлення структури програми у формі графової моделі, в якій кожна вершина представляє оператор, а дуга — передачу керування між операторами. На основі графової моделі визначається, чи досяжні вершини програми і чи є вихід з усіх потоків керування програми для її завершення. Цей підхід дозволяє виявити логічні помилки.

Для проведення аналізу потоків даних зазначений граф розширюється показниками змінних, їхніми значеннями і посиланнями для кожного оператора програми. Під час тестування потоків даних спочатку визначаються значення предикатів в операторах реалізації логічних рішень, за якими формуються шляхи виконання програми. Потім проводиться перевірка обчислень на арифметичних операціях. Для простежування шляхів встановлюються точки, в яких є посилання на змінну до присвоєння їй значення, або змінній присвоюється значення без її опису, або наводиться повторний опис змінної, або до змінної немає звернення.

**Методи доведення коректності програм.** Підхід до доведення коректності було сформульовано Флойдом і Науром, а потім розвинуто в роботах Хоара [10, 14].

Розвиток цього підходу в 70-ті роки показав труднощі його практичного застосування для значних за обсягом програмних систем, тому що опис процесу доведення коректності програми значно перевершує опис самої програми. Крім того, для доведення потрібні деякі логічні твердження введення—виведення, щодо яких може бути перевірено реальні функціональні можливості даної програми.

Формування таких тверджень виявилось дуже складним завданням, особливо для програм з високим ступенем паралелізму і взаємодії з користувачем. Крім того, важко перевірити достатність цих тверджень.

Спочатку доведення коректності застосовувалося до вже написаних програм. У 80-ті роки цей метод було поширено на доведення коректності створюваних програм, як і метод дисциплінуючої розробки. За даним методом, використовуючи отримані вхідні й вихідні твердження, програміст послідовно декомпонує задачу на кілька підзадач, визначаючи для кожного твердження умови та висновки до та-

кого рівня, коли текст програми можна буде безпосередньо включити між вхідними і вихідними твердженнями.

За такого підходу число помилок значно менше, ніж при традиційному. Це забезпечує не лише скорочення часу на тестування, а й сприяє досягненню повноти, однозначності і несуперечності специфікацій.

**Метод аналізу дерева відмов.** Цей метод прийшов у програмну інженерію з техніки, в якій він широко застосовується для аналізу несправностей апаратури. Суть методу полягає у виборі “відмовної ситуації” в окремій компоненті системи, простежуванні подійних ланцюжків, що могли б призвести до її виникнення, та побудові дерева відмов, що використовує логічні зв’язки *I*, *АБО*. З іншого боку, проглядається вплив відмови однієї компоненти на програмне забезпечення в цілому. У такий спосіб будується дерево відмов. Цей метод застосовується як на модульному рівні, так і для аналізу функціонування комплексу. Відомий досвід його успішного застосування у розробці систем реального часу.

Аналіз програми методом дерева відмов пов’язаний із доведенням коректності і допомагає “організувати” подробиці доведення, зокрема в системах реального часу. Метод не залежить від специфікацій програми.

**Метод аналізу інтерфейсів.** Застосовується для контролю передачі параметрів між різноманітними об’єктами і включає перевірку:

- відповідності числа аргументів, переданих об’єктам, числу їхніх параметрів;
- відповідності атрибутів параметрів та аргументів і порядку їхнього розміщення;
- правильності задання числа аргументів, їхніх атрибутів і порядку проходження у викликах функцій;
- узгодженості означування зовнішніх змінних в усіх використовуваних об’єктах;
- передача констант як аргументів тощо.

**Метод перевірки несуперечності.** Застосовується в аналізі логіки програми для виявлення операторів, які ніколи не використовуються, а також для виявлення суперечностей у логіці програми. Цей метод часто називають методом аналізу потоків керування вхідних даних, частину яких подано в символічному вигляді. Результатом виконання є значення змінних, виражених формулами над вхідними даними.

Розрізняють два види задач.

1. Побудова тестового набору даних для заданого шляху, що визначає цей шлях при символічному виконанні. Якщо такого набору немає, робиться висновок про нереалізованість даного шляху;

2. Визначення шляху, який буде пройдено при заданих обмеженнях на вхідні дані як деяких областей значень вхідних об’єктів, і

задання символічних значень змінних наприкінці шляху, тобто побудова функції, котра реалізує відображення вхідних даних у вихідні.

Розглянемо ці два види задач символічного виконання.

1. Нехай  $P(X, Y)$  — програма, що виконується над символами на деяких наборах даних,  $D = (d_1, d_2, \dots, d_n)$ ,  $X$  — множина вхідних даних  $P$ ,  $Y$  — множина вихідних даних,  $D \subset X$ .

Пронумеруємо оператори програми  $P$ .

Стан виконання програми — це трійка  $\langle N, pc, Z \rangle$ , де  $N$  — номер поточного оператора програми  $P$ ,  $pc$  (part condition) — умова вибору шляху в програмі (спочатку це істина), що є логічним виразом над  $D$ ;  $Z$  — множина пар  $\{ \langle z_i, e_i \rangle \mid z_i \in X \cap Y \}$ , у яких  $z_i$  — змінні програми,  $e_i$  — їхні значення, а  $i = 1, \dots, K$ ;  $Y$  — множина вихідних даних.

Семантика символічного виконання задається правилами оперування символічними значеннями, відповідно до яких арифметичні обчислення замінюються алгебраїчними. Задамо семантику символічного виконання через базові конструкції мов програмування. Для імперативних мов базовими конструкціями є оператори присвоювання, переходу й умовні оператори.

В операторі присвоювання  $Z = e(x, y)$ ,  $x \in X$ ,  $y \in Y$ , у виразі  $e(x, y)$  провадиться підстановка символічних значень змінних  $x$  і  $y$ , унаслідок чого утворюється вираз  $e(D)$ , що стає значенням змінної  $z$  ( $z \in x \cup y$ ). Вхідження в отриманий вираз  $e(D)$  змінних з  $Y$  означає, що їхніх значень не визначено і значення  $z$  не визначено.

За допомогою оператора переходу керування передається оператору, позначеному відповідною міткою.

В умовному операторі “якщо  $\alpha(x, y)$  то  $B1$  інакше  $B2$ ” обчислюється вираз  $\alpha(x, y)$ . Якщо його визначено і він дорівнює  $\alpha'(D)$ , то утворюються логічні формули:

$$pc \rightarrow \alpha'(D); \quad (1)$$

$$pc \rightarrow \neg \alpha'(D). \quad (2)$$

Якщо  $pc$  — хибне (false), то лише одна з останніх формул може бути виконаною. Тоді:

— якщо здійснення (1), то керування передається на  $B1$ ;

— якщо здійснення (2), то керування передається на  $B2$ ;

— якщо (1), (2) не здійсненні (тобто з  $pc$  не випливає ні  $\alpha'(D)$ , ні  $\neg \alpha'(D)$ ), тоді є принаймні один набір даних, що задовольняє  $pc$  і відповідає частині “то” умовного оператора, а також є набір даних, відповідний “інакше” цього умовного оператора.

Отже, утворюються два шляхи символічного виконання, яким відповідають свої  $pc$ :

$$pc1 = pc \wedge \alpha' (D);$$

$$pc2 = pc \wedge \neg \alpha' (D).$$

Символьне виконання, націлене на побудову тестового набору даних, реалізується таким алгоритмом:

- припускаємо, що  $pc = \text{істина} (\text{true})$ ;
- для заданого шляху формуємо  $pc$  відповідно до семантики операторів, для умовного оператора семантику трактуємо як перетворення  $pc$  виду (1) або (2);
- розв'язуємо обидві формули рівнянь (1) і (2).

Розв'язання цих систем дає тест для перевірки шляхів, коли ж таких розв'язань немає, то це означає невиконання шляху.

2. Визначення шляху при заданих обмеженнях на вхідні дані проводиться в такий спосіб:

- припускаємо  $pc = \beta (D)$ , де  $\beta (D)$  — вхідна специфікація, коли ж її немає, то  $pc = \text{істина}$ ;
- здійснюємо символічне виконання відповідно до семантики операторів, якщо трапляється розгалуження, то запам'ятовується стан у даній точці або вибирається одна з гілок; вибирається нова специфікація і виконується умовний оператор, при якому формується стан програми при умові  $pc$ ;
- наприкінці шляху сукупність станів вихідних змінних описує функцію програми і набір даних, що задовольняє  $pc$  і є тестом, який покриває даний шлях;
- для всіх проміжних  $\delta (x, y)$  з вихідною специфікацією  $\gamma (x, y)$  робиться спроба довести можливість виконання таких логічних формул:

$$pc \rightarrow \delta (x, y) \quad \text{і} \quad pc \rightarrow \gamma (x, y),$$

де  $pc$  є значенням поточної умови в даній точці програми.

Для доведення цих формул треба провести верифікацію програми на даному шляху.

Для встановлення шляху вирази декомпонуються на множину нерівностей. Якщо ця множина містить якісь несумісності, то шлях не можна встановити. У разі сумісності множини утвориться множина даних, що будуть використовувати даний шлях під час символічного виконання.

**Інші методи.** Методи імітаційного моделювання застосовуються, наприклад, для встановлення функціональної адекватності і прийнятності розроблювального програмного забезпечення до початку розробки. При імітації специфікацій перевіряється їхня повнота і несуперечність, для чого їх потрібно виразити певною формальною мовою. Це є єдиний метод тестування специфікацій. Крім цього, імітація може використовуватися і для одержання вимушених обмежень

розробки, перевірки відповідності створюваної програми її умовам, оцінки продуктивності. Для імітації поведінки системи на етапі проектування застосовуються мережові моделі, зокрема мережі Петрі, функціональні діаграми та ін.

Символьне тестування (виконання) застосовується для формування тестових даних з метою перевірки окремих шляхів з використанням символічних значень замість реальних: вхідні значення позначаються символами, й оператори виконуються з використанням елементарної алгебри. Виконання визначеного шляху звичайно включає умовні переходи, символічні вирази, до яких входять як алгебраїчні, так і булеві конструкції.

З метою проведення статичного аналізу застосовуються інструменти, котрі служать для дослідження структури програми і визначення деяких видів помилок у програмному забезпеченні: нездійсненні коди, неініційовані змінні, які намагаються використати, ініційовані, але не використововані змінні тощо. Крім того, розробляється велика кількість засобів автоматизації символічного виконання програм, наприклад, автоматизований засіб статичного контролю розробки й інструменти автоматизації доведення коректності. З метою перевірки специфікацій систем для паралельних обчислень використовується автоматизований апарат мереж Петрі.

### 7.1.2. Динамічні методи тестування

Під динамічним тестуванням розуміють перевірку коректності й надійності розроблювального програмного забезпечення за допомогою його виконання на ЕОМ.

Динамічне тестування ґрунтується на систематичних, статистичних (ймовірнісних) та імітаційних методах.

Дамо стисло характеристику цих методів.

Систематичні методи тестування поділяються на методи, в яких програми розглядаються як “чорна скринька” (використовується інформація про розв'язувану задачу), і методи, в яких програма розглядається як “біла скринька” (використовується інформація про структуру програми).

Тестування програм за принципом “чорної скриньки” називають тестуванням за даними шляхом керування за входами — виходами. Мета такого тестування — з'ясувати обставини, при яких поведінка програми не відповідає її специфікації. При цьому кількість виявлених помилок у програмі є критерієм якості вхідного тестування. Цього можна досягнути, якщо в ролі тестових наборів буде використано всі можливі набори вхідних даних, що неможливо через величезну (в загальному) кількість варіантів вхідних даних.

Метою динамічного тестування програм за принципом “чорної скриньки” стає виявлення одним тестом максимального числа поми-

лок, виявлених одним тестом, а тестування програми обмежується використанням невеликої підмножини всіх можливих вхідних даних.

**Методи “чорної скриньки”.** Розрізняють такі види динамічного тестування за принципом “чорної скриньки” [14]:

- еквівалентна розбивка;
- аналіз граничних значень;
- застосування функціональних діаграм, які в поєднанні з реверсивним аналізом дають досить повну інформацію про функціонування програм, що тестуються.

Еквівалентна розбивка полягає в розбивці вхідної області даних програми на скінченне число класів еквівалентності так, що кожний тест, який є представником певного класу, буде еквівалентний будь-якому іншому тесту цього класу.

Класи еквівалентності виділяються шляхом перебору вхідних умов і розбивки їх на дві або більше групи. При цьому розрізняють два типи класів еквівалентності: правильні, що подають вхідні дані програми, і неправильні, що подають помилкові вхідні значення.

Наприклад, якщо вхідна умова описує область значень даного як ціле, яке може приймати значення від 1 до 999, то визначається один правильний клас еквівалентності “ $1 < \text{значення цілого даного} < 999$ ” та два неправильних “значення цілого даного  $< 1$ ” і “значення цілого даного  $> 999$ ”.

Розробка тестів методом еквівалентної розбивки здійснюється за два етапи: виділення класів еквівалентності і побудова тестів. Доцільно використовувати цей процес і до вихідної області програми. У цьому разі в побудові тестів (вибір вхідних даних) пропонується застосовувати символічне виконання програм, що переслідує іншу мету (визначення символічних значень наприкінці шляху), але спрямоване у зворотний бік, тобто, виходячи з обмежень на вхідні дані, програма відновляються символічно з її кінця.

Таким способом наприкінці зворотного символічного виконання (на початку програми) можна одержати обмеження на вхідні дані, при яких програма виконувалася б у помилковому напрямі. Потім відповідно до обмежень генеруються тести. Цей підхід дозволяє цілеспрямовано розбивати вхідну область даних на еквівалентні класи відповідно до потенційно допустимих помилок на виході програми.

Отже, методи тестування за принципом “чорної скриньки” застосовуються для тестування реалізованих у програмі функцій шляхом перевірки невідповідності між реальною поведінкою функцій та очікуваною поведінкою відповідно до початкових специфікацій вимог. Під час підготовки до цього виду тестування будуються таблиці умов, графи причин—наслідків та області розбивки. Крім того, готуються набори тестів, які враховують параметри й умови середовища, що впливають на поведінку функцій, а також для кожної умови

визначаються множини значень і обмежень на число предикатів, що тестуються.

**Методи “білої скриньки”.** Стратегія “білої скриньки” дозволяє досліджувати внутрішню структуру програми, причому виявлення всіх помилок у програмі є критерієм вичерпного тестування маршрутів її потоків (графу) передач керування.

Оскільки вичерпне тестування шляхів (цей критерій відповідає повному структурному комбінаторному тесту) виключається, на практиці застосовуються слабкіші критерії:

(а) — критерій покриття операторів — набір тестів у сукупності має забезпечити проходження кожного оператора не менше одного разу;

(б) — критерій тестування гілок (відомий як покриття рішень або покриття переходів) — набір тестів у сукупності має забезпечити проходження кожної гілки (кожного виходу оператора) принаймні один раз.

Критерій (б) відповідає простому структурному тесту і на практиці поширений більше. Щоб задовольнити цей критерій, необхідно побудувати систему шляхів, яка містить усі гілки програми. Знаходження такого оптимального покриття в деяких випадках здійснюється просто, але здебільшого є складним завданням.

Методи тестування за принципом “білої скриньки” орієнтовані на перевірку проходження всіх шляхів програми, що тестується, і включають шляхове та мутаційне тестування.

Шляхове тестування застосовується на рівні модулів і для тестування невеликих програм. В його основі лежить графова модель програми, що використовується для вибору і створення тестових ситуацій та тестових даних і включає тестування:

- операторів, які має бути виконано хоча б один раз. У цьому разі тестування не враховує помилок (наприклад, у разі передачі керування за неправильною адресою), тому помилки залишаються, бо програма містить набагато більше логічних шляхів, ніж операторів. Виконання всіх операторів забезпечує проходження лише підмножини логічних шляхів;

- шляхів за відповідним графом потоку керування для виявлення маршрутів передачі керування за допомогою так званих шляхових предикатів, до кожного з яких входить набір рівностей і нерівностей. Для обчислення предикатів створюється набір тестових даних, котрі гарантують тестування всіх шляхів. Усі шляхи виконати неможливо, тому деякі шляхи залишаються невиконаними і залишаються невиявленими помилки, котрі можуть виявитися в процесі експлуатації;

- блоків, які розділяють програми на окремі частини і які виконуються принаймні один раз, а в складних комбінаціях багаторазово, за допомогою знаходження шляху в програмі, що включає зазначене

сполучення блоків (наприклад тих, котрі реалізують одну функцію) або знаходження вхідної множини даних, що буде використовуватися для виконання зазначеного шляху.

Отже, коли ставиться питання, як тестувати готовий продукт, то, як правило, розглядають різноманітні підходи: структурну, “чорну” (закриту) або “білу” (відкриту) “скриньку”. Як було сказано раніше, методи структурної, відкритої, або “білої скриньки”, базуються на структурі вихідної програми, за якою будуються різноманітні тести, що охоплюють майже всі шляхи виконання програми.

Закрита, або “чорна скринька”, навпаки, не базується на структурі програми, тому що це машинний код і про його структуру нічого не відомо. Щодо цієї “скриньки” відомими є виконувані ними функції, входи (вхідні дані) і виходи (вихідні дані), а також логіка обробки в загальних рисах.

### 7.1.3. Функціональне тестування

Метод “чорної скриньки” застосовується для тестування функцій і належить до функціонального тестування. Його метою є виявлення невідповідностей між реальною поведінкою реалізованих функцій і очікуваною поведінкою відповідно до специфікації та вихідних вимог. Функціональні тести мають охоплювати всі функції, реалізовані в програмному забезпеченні, з урахуванням найбільш імовірних типів помилок. Тестові сценарії, котрі об’єднують окремі тести, орієнтовані на перевірку якості розв’язання функціональних задач.

Функціональні тести створюються за зовнішніми специфікаціями функцій, проектною інформацією та за текстом мовою програмування. Тести, основані на зовнішніх специфікаціях програмного забезпечення, належать до функціональних характеристик і застосовуються на етапі комплексного тестування й іспитів для визначення повноти розв’язання функціональних задач та їхньої відповідності вихідним вимогам. Функціональному тестуванню передують аналіз функцій, до завдань якого входять:

- ідентифікація множини функціональних вимог;
- ідентифікація зовнішніх функцій у реалізації програмного забезпечення і побудова послідовностей функцій відповідно до використання їх в програмному забезпеченні;
- ідентифікація множини вхідних даних кожної функції і визначення областей зміни їх;
- побудова тестових наборів і сценаріїв тестування функцій;
- виявлення й представлення всіх функціональних вимог за допомогою тестових наборів і проведення тестування помилок у програмі і у взаємодії із середовищем.

Тести з проектною інформацією пов’язані із структурами даних, алгоритмами, інтерфейсами між окремими компонентами і застосовують-

ся для тестування окремих компонентів та інтерфейсів між ними. Основна мета тестування — забезпечення повноти й узгодженості реалізованих у програмних компонентах функцій та інтерфейсів між ними.

Комбінований метод “чорної скриньки” і “білої скриньки” оснований на розбивці вхідної області функції на підобласті виявлення помилок. Підобласть містить однорідні елементи, які всі обробляються коректно або некоректно. Для тестування підобласті провадиться виконання програми на одному з елементів цієї області.

Підобласть доповнюється категоріями помилок, що дає можливість зосередити пошук імовірних помилок на множині входів функції. Методологія пошуку ймовірних помилок використовує специфікації і вихідний код. Тестувач спочатку виконує розбивку за специфікаціями, переглядаючи класи входів, що обробляються однакою спосібом. Потім створюється розбивка програми на послідовності шляхів, що також обробляються однакою.

Розбивка утворюється перетином проблемної розбивки і шляхової розбивки для створення множини класів еквівалентності, елементи яких трактуються однакою. Множина тестів створюється шляхом вибору одного елемента з кожного класу розбивки.

Таким чином, передумовами тестування є:

- коректне формування вимог та обмежень на якість програмного забезпечення;
- коректний опис моделі функціонування програмного забезпечення в середовищі експлуатації його замовником;
- адекватність моделі програмного забезпечення заданому класу.

До цього часу було розроблено автоматичні засоби підтримки тестування. Вони здійснюють аналіз повноти тестів, трасування потоків шляхів, а також перевірку (контроль) тверджень. Твердження — це речення, що встановлює якийсь факт. Іншими словами, це встановлення істинності факту в процесі виконання програми.

Твердження можуть бути “локальними” і “глобальними”. “Локальне” твердження має бути істинним у точці його опису, а “глобальне” — в будь-якій точці програми. Контроль тверджень полягає в оцінці тверджень під час виконання програми і реєстрації їхньої істинності або хибності.

Аналізатор повноти тесту — це програма, яка, будучи застосованою до іншої, вже “аналізованої” програми, забезпечує облік виконань кожного логічного сегмента цієї програми.

Спочатку програма, що аналізується, проглядається і розподіляється на досить малі сегменти. Звичайно, цей розподіл виконується на рівні логічних переходів. Кожний логічний перехід у програмі є початком двох або більше таких сегментів.

Після виконання кожного тестового прикладу аналізатор видає таблицю про виконання кожного сегмента. Під час виконання кож-

ного тестового прикладу утворюється нова таблиця сегментів, яка показує число виконань сегментів для цього тесту.

Якщо після виконання всіх тестових прикладів знайдуться один або більше сегментів, для яких число виконань дорівнює 0, то, вочевидь, є неперевірена частина програми.

## 7.2. Методи доведення правильності програм

Роботи з доведення правильності програм появились ще у 80-ті роки [10,11,13]. Найвідоміші методи доведення програм:

- метод рекурсивної індукції, індуктивних тверджень Флойда — Наура;
- метод структурної індукції Хоара;
- метод Дейкстри тощо.

**Метод Флойда — Наура** — неформальний метод індуктивних тверджень. Суть методу полягає в такому: визначаються умови на вхідні і вихідні дані, вибираються контрольні точки програми в такий спосіб, щоб будь-який циклічний шлях проходив хоча б через одну таку точку; для точок формулюються твердження про стан змінних у цих точках (для циклів ці твердження мають бути істинними під час кожного проходження циклу — інваріанти циклу).

Кожна точка розглядається, як індуктивне твердження (формула, яка залишається істинною для кожного звернення в цю точку програми і залежить не лише від вхідних та вихідних, а й від проміжних змінних). Далі за індуктивними твердженнями й умовами на аргументи та результати програми будуються умови правильності цієї програми.

Потім для кожного шляху програми між двома точками ставляться у відповідність умови правильності і доводиться істинність усіх цих умов, а також завершення програми за даними, які задовольняють вхідні умови. Доведення правильності програми впливає із доведення істинності умов правильності.

**Метод Хоара** [10] — це удосконалений метод Флойда — Наура, що базується на аксіоматичному описі семантики мови програмування, в якому кожна аксіома виражає зміну значень змінних за допомогою операторів цієї мови. У цьому методі основну увагу приділено формалізації операторів переходу й виклику процедур за допомогою правил виведення, що мають індуктивне висловлення для кожної мітки та функції вихідної програми.

Оператор переходу розглядається як засіб виходу з глибоких циклів та аварійних виходів. Потім Хоар розвинув метод індуктивних висловлень шляхом введення до них необмежених викликів процедур з деякими обмеженнями на параметри процедур. Ця ідея розвивається в напрямі доповнення системи правил виведення з ви-

користанням перейменувань глобальних змінних. Визначена система правил виведення відповідає виводу програми з умовами на аргументи та результати, а також на правильність цієї програми.

Опис доведення при цьому є досить громіздким і складним, а також відрізняється неповнотою і ненадійністю, тому для нього не було розроблено автоматизованих засобів у повному обсязі.

**Метод рекурсивних індукцій Дж. Маккарті** [10], котрий пізніше розвинув Р. Бурстал [11], полягає в структурній перевірці властивостей функцій, що обробляють структурні типи даних. Для доведення правильності програм застосовуються методи, які змінюють структури даних, діаграми переходу Флойда та метод Манна, що полягає в доведенні за допомогою набору фактів під час символічного виконання програм.

**Метод Дейкстри** [15] включає два підходи до доведення правильності програм. Перший оснований на моделі обчислень, які оперують історіями обчислень, що виникають у роботі програм. Цей підхід вимагає обробки великого обсягу інформації під час аналізу шляхів виконання в моделі обчислень.

Другий підхід базується на формальному дослідженні тексту програми за допомогою предикатів першого порядку. Він орієнтований на клас асинхронних програм, в яких виникають проблеми зі збереженням станів програм під час виконання їх.

При взаємодії послідовних процесів використовується механізм “збирача сміття”. Основна ідея доведення правильності програм за методом Дейкстри полягає у привнесенні в процес написання програм таких методів, як переобчислення, математична індукція й абстракція.

Метод переобчислення базується на інваріантних відношеннях, якими перевіряються межі обчислювань, і дозволяє перевіряти правильність обчислювань.

Математична індукція застосовується для організації представлення циклу та рекурсивних процедур за допомогою необхідних і достатніх умов повторення обчислень.

Абстракція дозволяє послабити кількісні обмеження, що накладаються на дослідження програми за допомогою переобчислень.

**Метод Р. Андерсена** [14] базується на припущенні, що правильність програм можна доводити як теореми в математиці. В основі методу доведення лежить апарат математичної індукції, суть якого полягає у неформальному доведенні правильності, внаслідок чого можуть допускатися помилки доведення, до того ж не всі помилки в програмі піддаються виявленню. Цей метод слід розглядати як системний спосіб перевірки правильності програми за столом.

**Метод математичної індукції** доведення правильності програм дозволяє довести істинність деякого припущення  $P(n)$ , що зале-

жить від параметра — цілого числа  $n$ , для всіх  $n \geq n_0$ , починаючи з доведення істинності  $P(n_0)$ . Потім, припускаючи істинність  $P(n-1)$  для значення  $n-1$ , доводиться істинність  $P(n)$ . Цього достатньо, щоб довести істинність  $P(n)$  для всіх  $n \geq n_0$ .

Схема застосування методу полягає в такому.

Нехай потрібно довести справедливості деякого твердження  $A$ , коли виконання програми досягає визначеної точки. Проходячи через цю точку  $n$  разів, можемо визначити справедливості  $A(n)$ , якщо доведемо:

1) що справедливе  $A(1)$  (тобто справедливе висловлення  $A$  при першому проході через задану точку);

2) якщо справедливе  $A(n-1)$  при  $(n-1)$ -му проходженні через задану точку, то справедливе й  $A(n)$  за умови потрапляння в задану точку  $n$  разів.

Для доведення правильності програми необхідно описати її функціонування у формі висловлень про правильність або просто тверджень. Якщо припущення, що працююча програма зрештою завершиться, то після її закінчення справедливість твердження  $A$  означає правильність програми.

Для доведення правильності програми застосовувалися теоретико-множинні методи, методи верифікації створюваних програм на етапах життєвого циклу, а також тестування отриманих програм на множині тестових даних для встановлення їхньої правильності. Треті широко використовуються і нині.

### 7.2.1. Техніка формального доведення

Формальне доведення програм полягає в конвертуванні коду програми до його логічної структури за допомогою таких кроків.

1. Описується твердження (висловлення), що задає входи — виходи компонентів за допомогою операторів, які являють собою комбінацію логічних змінних (true/false), пов'язаних між собою операціями, наведеними в таблиці 1.

Таблиця 1. Логічні операції

Назва	Приклади	Значення
Кон'юнкція	$x \& y$	$x$ і $y$
Диз'юнкція	$x * y$	$x$ або $y$
Заперечення	$\neg x$	не $x$
Імплікація	$x \rightarrow y$	якщо $x$ , то $y$
Еквівалентність	$x = y$	$x$ рівнозначно $y$
Квантор загальності	$\forall x P(x)$	для всіх $x$ умова істинна
Квантор існування	$\exists x P(x)$	існує $x$ , для якого $P(x)$ істинна

Приклад. Нехай програмна компонента приймає на вході одини-мірний масив цілих чисел  $T$  довжини  $N$  (тобто  $T[1 : N]$ ). На основі цього масиву потрібно створити еквівалентний масив  $T'$  тієї довжини  $N$ , що містить елементи  $T$  в порядку зростання їхніх величин.

Вхідні умови можна записати як твердження  $A_{beg}$ :

$(T[1 : N] \text{ — масив цілих}) \& (T'[1 : N] \text{ масив цілих})$ .

Аналогічно записуються вихідні твердження  $A_{end}$  у вигляді кон'юнкції таких умов:

(а)  $(T \text{ — масив цілих}) \& (T' \text{ — масив цілих})$

(б)  $(\forall i \text{ якщо } i \leq N \text{ то } \exists j (T'(i) \leq T'(j)))$

(в)  $(\forall i \text{ якщо } i < N \text{ то } (T'(i) \leq T'(i+1)))$ ,

тобто  $A_{end}$  — це:

$(T \text{ — масив цілих}) \& (T' \text{ — масив цілих})$

$\& \forall i \text{ якщо } i \leq N \text{ то } \exists j (T'(i) \leq T'(j))$

$\& (\forall i \text{ якщо } i < N \text{ то } (T'(i) \leq T'(i+1)))$ .

2. Для переупорядкування елементів масиву  $T$  в порядку зростання їхніх величин використовується пазиркове сортування, що виконується в масиві  $T'$ , в який попередньо копіюється масив  $T$ .

Алгоритм сортування представлено на рис. 7.1. Операторів алгоритму розміщено в прямокутниках. Умови, за якими відбувається вибір альтернативних шляхів в алгоритмі, представлено в паралелограмах. Кружками відзначено точки з початковою  $A_{beg}$  і кінцевою умовами  $A_{end}$  та станами алгоритму: кружок з нулем — початковий стан, кружок з однією зірочкою позначає стан після обміну місцями двох сусідніх елементів масиву  $T'$ , кружок з двома зірочками позначає стан після обміну місцями всіх пар за один прохід через весь масив. Крім уже відомих змінних  $T$ ,  $T'$  та  $N$ , в алгоритмі використано ще дві змінні:  $i$  типу ціла та  $M$  — булева, значенням якої є логічні константи true та false.

Для доведення того, що алгоритм дійсно забезпечує виконання вихідних умов, розглянемо послідовно динаміку виконання цих умов у визначених точках алгоритму. Зауважимо, що вказані точки ділять алгоритм на відповідні частини, правильність кожної з яких обґрунтовується окремо.

Так, оператор присвоювання означає для всіх  $i$  таких, що  $(i \leq N) \& i > 0$  виконується  $(T'[i] := T[i])$ .

Результат виконання в точці з нулем може бути виражено твердженням (теоремою):

$(T[1 : N] \text{ — масив цілих}) \& (T'[1 : N] \text{ — масив цілих})$

$\& (\forall i \text{ якщо } i \leq N (T'[i] = T[i]))$ .

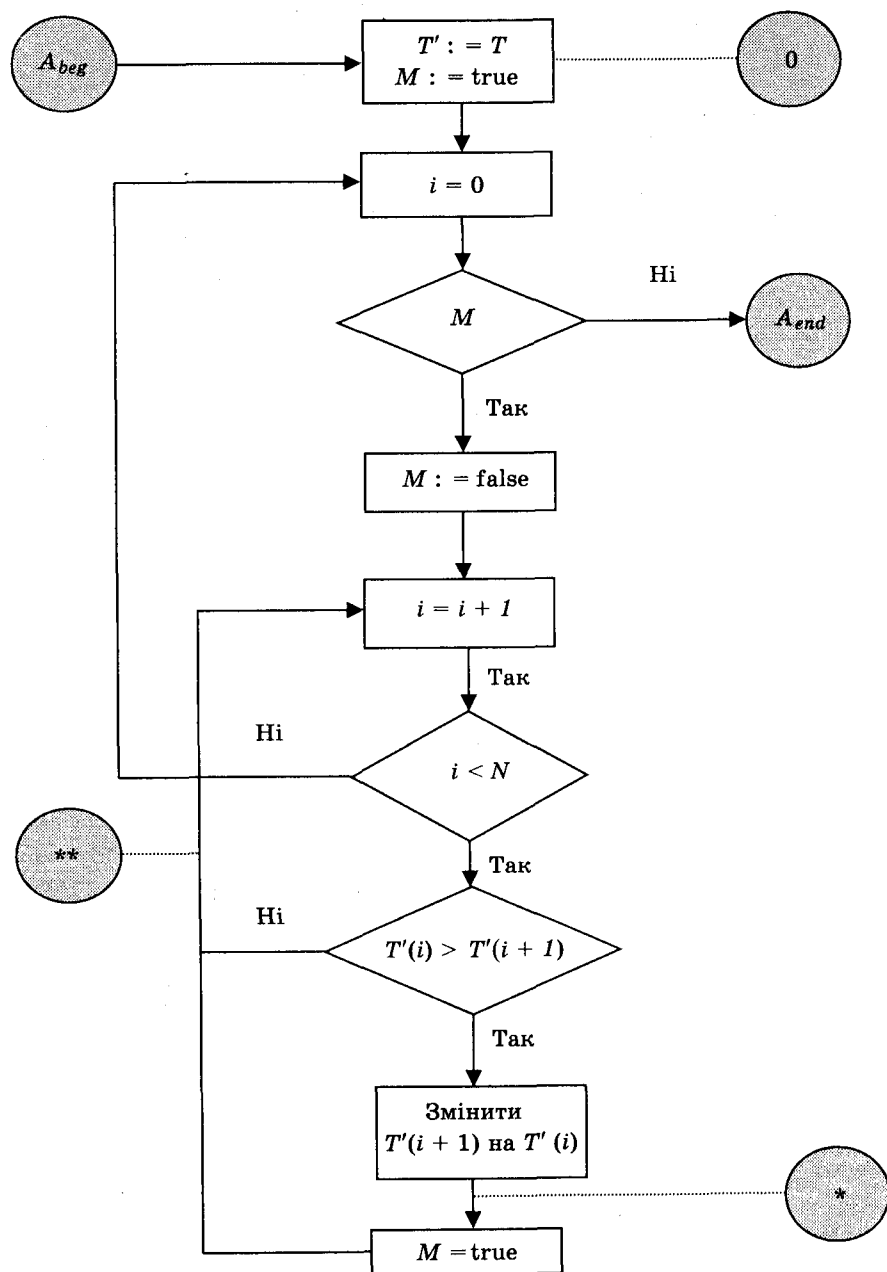


Рис. 7.1. Схема переупорядкування елементів масиву

Доведення очевидне, оскільки за семантикою оператора присвоювання (поелементне пересилання чисел із  $T$  в  $T'$ ) самі елементи при цьому не змінюються, до того ж у даній точці порядок їх в  $T$  і  $T'$  однаковий. Отже, виконується умова (б) вихідного твердження.

Зауважимо, що перший рядок доведеного твердження збігається з умовою (а) вихідного твердження  $A_{end}$ , яка залишається справедливою до кінця роботи алгоритму, а тому в наступних твердженнях ми її наводити не будемо.

В точці з однією зірочкою виконано оператор

$(i < N) (T'(i)) > T'(i + 1) \rightarrow (T'(i) \text{ та } T'(i + 1) \text{ — міняються місцями.})$

Унаслідок роботи оператора стає справедливим таке твердження:

$\exists i$  якщо  $i < N$  то  $T'(i) < T'(i + 1)$ , яке є частиною умови (в) твердження  $A_{end}$  (для однієї конкретної пари суміжних елементів масиву  $T'$ ).

Очевидно також, що семантика оператора обміну місцями не порушує умови (б) вихідного твердження  $A_{end}$ .

У точці з двома зірочками виконано всі можливі обміни місцями пар суміжних елементів масиву  $T'$  за один прохід через  $T'$ , тобто оператор обміну працював один або більше разів. Однак пузиркове сортування не дає гарантії, що досягнуто упорядкування за один прохід по масиву  $T'$ , оскільки після чергового обміну індекс  $i$  збільшується на 1 незалежно від того, як співвідносяться новий елемент  $T'(i)$  з попереднім елементом  $T'(i - 1)$ .

У цій точці також справедливе твердження:

$\exists i$  якщо  $i < N$  то  $T'(i) < T'(i + 1)$ .

Частина алгоритму, позначена точкою з двома зірочками, виконується доти, доки не буде упорядковано весь масив, тобто не буде виконуватися умова (в) твердження  $A_{end}$  для всіх елементів масиву  $T'$ :

$\forall i$  якщо  $i < N$  то  $T'(i) \leq T'(i + 1)$ .

Отже, виконання вихідних умов забезпечується відповідною семантикою операторів перетворення масиву та порядком виконання їх.

3. Із цього твердження генерується серія теорем, що доводяться. Починаючи з першого твердження і переходячи від одного перетворення до іншого, виробляється шлях доведення, який полягає в тому, що, якщо одне твердження є істинним, то є істинним й інше. Іншими словами, якщо перше твердження —  $A_1$  і перша точка перетворення —  $A_2$ , то першою теоремою є:

$$A_1 \rightarrow A_2.$$

Якщо  $A_3$  є такою точкою перетворення, то другою теоремою буде:

$$A_2 \rightarrow A_3.$$

Таким чином, формулюється загальна теорема:

$$A_i \rightarrow A_j.$$

де  $A_i$  і  $A_j$  — суміжні точки перетворення.

Остання теорема формулюється так, що умова “істинне” в останній точці відповідає істинності вихідного твердження:

$$A_k \rightarrow A_{end}.$$

Відповідно можна повернутися потім до точки перетворення  $A_{end}$  і до попередньої точки перетворення. Довели, що

$$A_k \rightarrow A_{end},$$

потім

$$A_j \rightarrow A_{j+1}$$

і так далі, доки не одержимо, що

$$A_1 \rightarrow A_0.$$

Результат будь-якого підходу до доведення програм буде такий самий.

4. Потім специфікуємо твердження if — then для кожного з них.

5. Щоб довести, що програма коректна, необхідно послідовно розташувати всі частини, що починаються з  $A_1$  і закінчуються  $A_{end}$ . Послідовність цих частин визначає, що істинність вхідної умови спричиняє істинність вихідної умови.

6. Після ідентифікації всіх частин необхідно перевірити істинність кожної частини, точно стверджуючи, що вхідні твердження мають своїм наслідком вихідні твердження, які відповідають перетворенням цих частин.

7. Доведено, що програма завершена.

**Переваги і недоліки доведення правильності.** При конструюванні ручного або автоматичного доведення виявляються помилки в

коді алгоритму. Техніка доведення додатково забезпечує формальне розуміння програми, оскільки перевіряється основна логічна структура. Регулярне використання цього підходу приводить до більш точного й строгого специфікування даних, структур даних й алгоритмічних правил.

Проте за точність треба платити. Багато хто відмовляється від доведення, тому що, наприклад, алгоритм пузиркового сортування набагато простіший, ніж його логічний опис і доведення. Крім того, великі і складні компоненти можуть включати логічні діаграми, перетворювачі та верифікацію великого числа частин.

Наприклад, програми для обробки нечислових даних можуть бути важчими для розуміння логіки, ніж для числових. Паралельну обробку також важко перевіряти, структури даних дають результат тільки після складного перетворення операторів для такого виконання.

Зауважимо, що техніка доведення базується лише на перевірці того, як вхідні твердження трансформуються у вихідні відповідно до логічних правил. Доведення коректності програми в логічному сенсі ще не означає, що в програмі немає помилок. Справді, ця техніка не розпізнає помилок у проєкті, в інтерфейсах з іншими компонентами, в інтерпретації специфікацій, у синтаксисі і семантиці мов програмування або в документації.

## 7.2.2. Техніка символічного виконання

Техніка логічного доведення ігнорує структуру і синтаксис мови програмування, в яких тестові програми виконуються. Ймовірний випадок, коли ця техніка доводить, що спроектовані компоненти є правильними, але вони не завжди виконуються. Інші техніки беруть до уваги характеристики мов.

Одна з подібних технік — *символьне виконання* — включає моделювання виконання коду, використовуючи символи замість змінних даних. Тестова програма розглядається як така, що має детермінований вхідний стан при вводі даних і умов. Завдяки тому, що кожний рядок коду виконується, техніка перевіряє, чи змінився стан. Кожний змінений стан зберігається, і виконувана програма розглядається як серія змін станів. Тобто, кожній логічній частині програми відповідає упорядкована серія змін станів. Останній стан кожної частини буде вихідним станом і програма буде правильною, якщо кожний можливий стан генерує вихідний стан.

Розглянемо символічне виконання на прикладі кількох рядків програми:

```
a = b + c;
if (a > d) task x ( );    // виконати завдання x
else task y ( );          // виконати завдання y
```

Інструмент символічного виконання буде відзначати, що умова  $a > d$  може бути істинною або ні. Оскільки виконання умовного коду включає специфічні сполучення значень  $a$  та  $d$ , то засіб символічного виконання запише два можливих значення для  $a$  та  $d$ , тобто розглядає два випадки. При цьому робиться поділ множини даних на два окремих еквівалентних класи або категорії, і код може розглядатися з погляду того, як він реагує на кожну категорію даних. Якщо розглядати тільки кожний з еквівалентних класів даних, поданих як символи, то зростає число випадків для доведення.

Розроблене доведення може стати довшим, ніж власне написаний код, і немає впевненості, що доведення правильності не містить помилок. Крім того, техніка має багато переваг над логічним доведенням теорем, бо покладається на уважне трасування умов програмних частин, що змінюються. Навіть якщо техніка автоматизована, великий і складний код може потребувати перевірки багатьох станів, частин і великого часу виконання процесу. При автоматичних засобах важко простежити за циклами. Оскільки в коді використовуються індекси і вказівники, то розбивка на еквівалентні класи стає важкою.

При доведенні коректності програми група тестувачів розглядає тільки код та його вхідні і вихідні умови. Програма розглядається в термінах класів даних і умов, описаних у проекті. У такий спосіб доведення може не потребувати виконання коду, а лише розуміння того, що робиться програмою.

Проте користувачів може не задовольнити такий підхід. Щоб продемонструвати їм, що програма власне працює, необхідно показати, як код виконується. У цьому разі тестування стає серією експериментів, результати яких в сумі дозволяють зрозуміти, як програма буде поводитися в заданій ситуації. Якщо доведення показує, як програма буде працювати в гіпотетичному середовищі, яке описане проектом і вимогами, то тестування подає інформацію про те, як програма працює в актуальному операційному середовищі.

### 7.2.3. Верифікація та валідація

Тестування — не єдиний спосіб досягнення правильності програм. Тепер у розробках прикладних програмних систем більше уваги приділяється таким методам перевірки правильності як валідація (validation) та верифікація (verification) [5, 7, 19].

*Верифікація* відповідає на запитання: “Чи коректно реалізовано проєкт?”

Метою процесу верифікації є перевірка відповідності реалізації системи до специфікацій результатів проєктування й опису компоненти. Областю застосування верифікації є життєвий цикл (ЖЦ) програмного забезпечення. Вона дозволяє відповісти на запитання: коректно реалізовано проєкт чи ні?

*Валідація* полягає в перевірці відповідності створеного програмного забезпечення потребам та вимогам замовника і потребує виконання таких завдань для отримання коректних програм та систем:

- планування процедур перевірки й контролю за методиками інспекцій проєктних рішень та нагляду за ходом розробки;
- підвищення рівня автоматизації проєктування програм з використанням CASE-систем [16];
- перевірка правильності функціонування програм за методами тестування на наборах цільових тестів;
- структурування системи на модулі, їхні окремі специфікації проєктування, реалізація й використання як повторних компонентів (reuse) [17];
- адаптація продукту до умов використання;
- керування проєктами.

Тобто, валідація — це підхід до перевірки на кожному етапі розробки програмного забезпечення на відповідність висунутим вимогам замовника до системи. Вона включає перегляди й інспекції як способи перевірки проміжних результатів на кожному етапі життєвого циклу з метою аналізу на відповідність і забезпечення якості функцій проєкту та створюваного продукту. Це дорогий процес, що забезпечує високу якість вихідного коду. Валідація дозволяє підтвердити, що програмне забезпечення є коректною реалізацією початкових умов у системі й провадиться по завершенню кожного етапу розробки цього забезпечення. Перегляди й інспекції результатів етапів проєктування та відповідності їх умовам замовника забезпечують якість створюваних окремих компонентів програмного забезпечення і системи в цілому.

Отже, основними завданнями верифікації і валідації є перевірка повноти, несуперечності й однозначності специфікацій вимог у створюваному програмному забезпеченні щодо сформульованих початкових вимог до системи.

**Елементи для перевірки.** Верифікація і валідація як підходи до перевірки правильності потребують планування цього процесу під час упорядкування вимог до системи з метою розподілу ресурсів і зосередження на найбільш критичних елементах проєкту, для яких коректність є чи не основною властивістю. До таких елементів відносять основні компоненти системи, а також:

- інтерфейси компонент системи (програмні, технічні й інформаційні) та взаємодії об'єктів (протоколів і повідомлень) для функціонування в сучасних розподілених середовищах;
- засоби доступу до баз даних і файлів (транзакції і повідомлення), що забезпечують захист від несанкціонованого доступу до них користувачів;
- документація до програмного забезпечення і системи в цілому;
- проєкти тестів, тестові процедури та вхідні дані;

— загальні та спеціальні засоби захисту інформації в системі.

Добре організовані способи перевірки на відповідність прийнятим вимогам на етапах проектування життєвого циклу і надання замовникові звітів про перевірку значною мірою впливають на коректність створюваного проекту системи. По закінченню проектування для зазначених елементів відповідно проводиться:

- перевірка правильності переведення проектів окремих компонент у вихідний код, а також описів їхніх інтерфейсів, трасування цих компонент і відповідних їм інтерфейсів зв'язку у зіставленні з вимогами замовника до взаємних зв'язків між функціями системи (до реалізації їх у комплексі);

- аналіз спроектованих засобів доступу до файлів або баз даних (як це сформульовано у вимогах) на відповідність принципам передачі даних, прийнятим у використовуваних системних засобах (СУБД, файлова система), процедурам маніпулювання даними і передача результатів перевірки компонентів;

- перевірка спроектованих засобів захисту, використовуваних відповідними компонентами, для задоволення вимог замовника до цих засобів і їхнє трасування на правильність реалізації та звернень до них з усіх точок системи.

Після перевірки всіх елементів системи проводяться комплексні роботи з об'єднання і валідації всієї системи в цілому. На цьому етапі спочатку тестують систему на множині підготовлених групою випробувань наборів тестів для визначення адекватності й достатності наборів даних та завершеності тестування.

Валідація проводиться до випробувань отриманої системи для перевірки її коректності. Аналізуються зауваження і помилки на проміжних етапах перевірки, способи усунення їх, а також проводиться перевірка правильності дотримання зовнішніх вимог замовника до всієї системи.

На етапі експлуатації системи до кола об'єктів перевірки належать: готовність персоналу, середовище для функціонування системи, документація контролю запуску й експлуатації системи тощо.

Верифікація і валідація мають спиратися на комплект методичних матеріалів і стандартів, що регламентують дії розробників, зокрема методики проектування та перевірки результатів, а саме:

- на збір даних і формування вимог на систему;
- на специфікацію елементів проекту (діаграми потоків даних і керування тощо);
- на інспекцію програм;
- на тестування на завершальному етапі;
- на випробування системи тощо.

**Методи верифікації і валідації.** Застосовуються методи статичного і динамічного аналізу. Методами статичного аналізу є інспекції,

аналіз потоків даних і керування, аналіз алгоритмів та обмежень, інтерфейсів і коду. До динамічного аналізу належать методи динамічного тестування під час виконання компонент і збору даних для оцінки повноти тестування. Функціональне тестування належить до методів динамічного тестування, проведеного на рівні автономного і комплексного випробувань працездатності системи.

### 7.3. Об'єкти тестування

Вид об'єкта тестування (компонента, група компонент, підсистема) визначає спосіб проведення тестування. Якщо вид об'єкта тестування належить до закритої або "чорної скриньки", компоненти якого невідомі, то тестування визначається вводом у нього вхідних даних для одержання вихідних даних. Мета тестування полягає в тому, щоб переконатися, що кожний аналізований вхідний набір даних відповідає очікуваному і простежуваному виведенню вихідних даних з машини.

При такому підході до тестування не потрібне знання внутрішньої структури і логіки об'єкта тестування. Проте повністю провести тестування в такий спосіб не завжди можливо. Покажемо це на простому прикладі програми розв'язання квадратного рівняння

$$ax^2 + bx + c = 0,$$

що припускає введення трьох чисел  $a, b, c$  і може видавати повідомлення типу "недійсний корінь". Тестувати повністю цю компоненту шляхом перебору всіх можливих триплетів  $a, b, c$  неможливо.

Для такої програми група тестування має вибрати представницькі тестові дані, щоб показати, що всілякі комбінації даних для програми в цілому подано. Наприклад, тестові дані може бути обрано так, щоб одержати всі комбінації позитивних, негативних чисел і 0 для кожного  $a, b, c$  (тобто 27 варіантів). Якщо відомо, як розв'язується рівняння знаходження квадратного кореня, то вибираються значення таких змінних, при яких дискримінант  $b^2 - 4ac$  існує для всіх класів трійок: позитивне, 0 або негативне. Проте, якщо тестові дані для кожного класу не викликають помилки, то це не дає гарантії того, що компонент вільний від помилок, тому що помилки можуть виникати при невідповідності типів даних або середовища оточення.

Для деяких типів об'єктів група тестування не може згенерувати представницьку множину тестових наборів, що демонстрували б функціональну правильність роботи компоненти при всіх їхніх можливих наборах тестів.

Іншим підходом до тестування об'єктів є відкрита програма або "біла скринька", при якому можна використовувати структуру об'єкта для організації тестування на різноманітних гілках. Наприклад,

можна виконати тестові набори, які проходять через всі оператори або всі контрольні точки компоненти, для того, щоб переконатися в правильності її роботи. Практично це можна зробити за допомогою такого підходу.

Нехай, наприклад, компонента з багатьма гілками і циклами має багато шляхів, що перевіряються. Навіть просту логічну структуру програми з ітерацією або рекурсією важко тестувати достовірно. Уявімо, що логіка компоненти структурована так, що цикл повторюється  $mn$  разів, як показано на рис. 7.2.

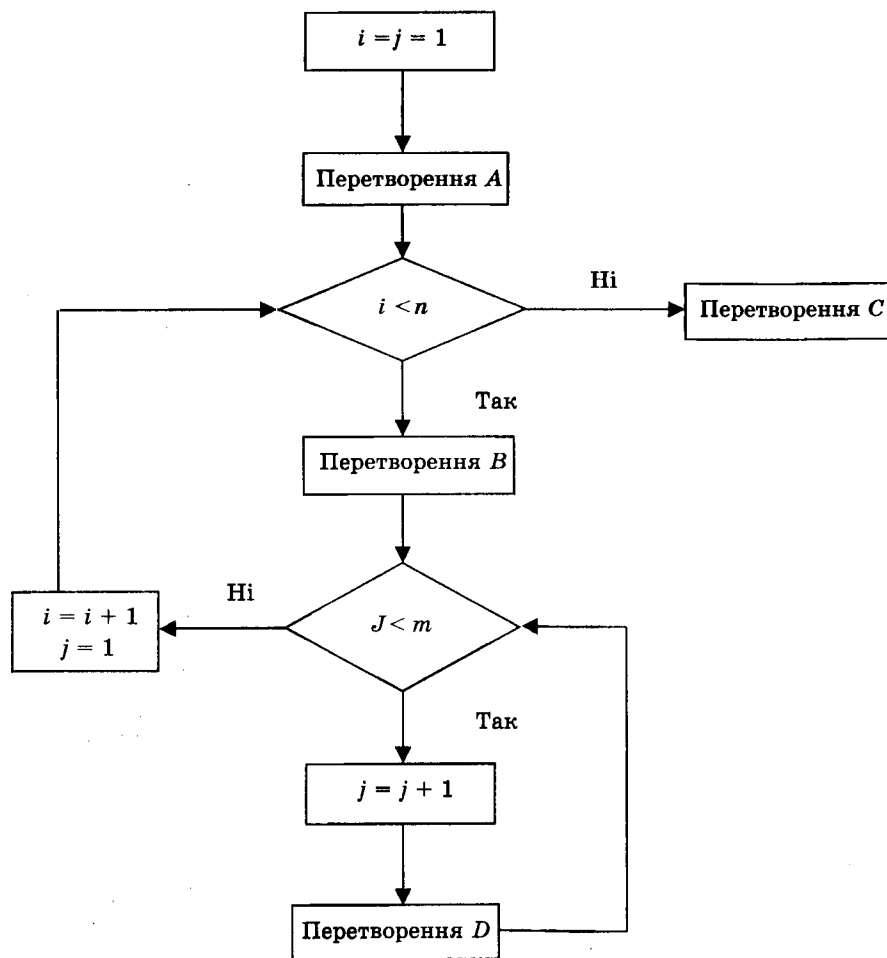


Рис. 7.2. Структура логіки для перевірки

Ця компонента має такі шляхи:

- перетворення  $A$  при  $i = 1, j = 1$ ;
- перетворення  $C$  при  $i \geq n$ ;
- перетворення  $B$  при  $i < n, j \geq m$ ;
- перетворення  $D$  при  $j < m$ .

Якщо кожний із  $n$  і  $m$  дорівнює 100 000, то цикл обробки тестового набору повториться 10 білйонів разів при проходженні всіх шляхів логіки. Можна використовувати стратегію тестування, при якій цикл буде виконуватися кілька разів, перевіряючи тільки мале число наборів із можливого числа.

Для цього прикладу можна вибрати значення  $i$  менше  $n$ , дорівнювало  $n$  або більше  $n$ ; аналогічно можна розглянути  $j$  менше  $m$ , дорівнює або більше  $m$ , а також як комбінацію всіх трьох значень  $i$ . У цілому стратегія може базуватися на даних, структурі, функції або на кількох інших критеріях.

Проектувальник тестів має заглянути всередину “чорної скриньки” і дослідити деталі процесів обробки даних, питання забезпечення захисту і відновлення даних, а також інтерфейси з іншими програмами та системами. Це сприяє підготовці тестових даних для проведення тестування.

Технологія тестування залежить від багатьох чинників:

- число можливих логічних шляхів;
- природа вхідних даних;
- кількість виконуваних обчислень;
- складність алгоритму.

### 7.3.1. Тестування окремого елемента

Мета тестування — перевірити, чи виконує розроблена окрема програма (система) задані функції, чи відповідають вони специфікаціям вимог і чи надійно програма працює. Для цього слід провести перевірку основної логіки програми шляхом контролю всіх її гілок (логічних шляхів) на комбінаціях вхідних даних за допомогою кількох видів стратегій тестування.

Гілка програми має пропрацювати хоча б один раз, щоб перевірити основні комбінації умов, включаючи помилкові ситуації.

До початкових етапів тестування належить налагодження програми. Його мета — знайти помилки в окремих елементах. Цей процес подібний до перевірки окремих одиниць (елементів) із класу. Спочатку перевіряється прочитаний вихідний код програми, відшукуються в ній синтаксичні помилки, а потім розпізнається алгоритм і дані. Код порівнюється із специфікацією в проекті і розглядаються всі релевантні випадки.

При компіляції програми знаходяться синтаксичні помилки. Коли пошук таких помилок завершено, розробляється тестовий набір для підтвердження того, що вхідні дані цілком перетворюються на очікувані вихідні дані. Ці кроки виконуються для кожного елемента тестування.

### 7.3.2. Перевірка коду

Опис проекту допомагає кодувати і документувати програму. Коментарі у вигляді слів і картинок допомагають пояснити, що робиться в рядках коду. Група експертів аналізує код і документацію для знаходження невідповідностей, несумісності та інших помилок в описі програми. Цей процес називається перевіркою коду і проводиться як перевірка відповідності результатів проектування розробленим вимогам. Технічними експертами можуть бути програмісти, проектувальники, проектувальники супервізорів та ін. До групи перевірки програми включаються програміст і три—чотири технічних експерти. Ця група перевіряє, чи задовольняються вимоги до проекту, їхню реалізацію та виконання відповідно до опису в документації.

Є два типи перегляду коду: статичний аналіз та інспекція.

При першому типі перегляду група перевірки визначає правильність коду і документації шляхом їхнього аналізу. Під час цього процесу проводиться неформальне обговорення і дискусія з приводу самого коду програми, а не її кодувальника. Головна увага приділяється знаходженню помилок, а не їхній фіксації.

### 7.3.3. Інспекції коду

Інспекція коду нагадує перший тип перевірки, однак є більш формальною. Вона орієнтована на перевірку опису, використовуваних типів і структур даних у проекті системи на їхню відповідність стандартам і процедурам. Група перевіряє алгоритми, засоби обчислень, а також їхню правильність та ефективність. Коментарі порівнюються на відповідність коду для визначення їхньої точності і повноти. Інтерфейси програмних елементів перевіряються на правильність на рівні їхнього опису.

Група експертів може також перевірити характеристики пам'яті або засоби обробки даних безпосередньо під час виконання на машині для порівняння оцінки з розробленими вимогами.

Коли група експертів перевіряє вихідний код виконавця, той, кого перевіряють, відчувається дуже незручно. Для успішного пошуку помилок до групи експертів включаються спеціалісти та досвідчені практики. Ідея полягає в тому, щоб якомога раніше знайти помилки в описах окремих компонент системи, тобто на етапі проектування, а не під час їхнього виконання. Навіть помилки в записі циклів легше знайти за столом, переглядаючи текст, ніж під час тестування.

Є приклад, коли група для проведення інспекцій, до складу якої входили три спеціалісти із середнім рівнем підготовки і три — чотири з високим, виявила більше десяти тисяч помилок на сторінках документації проекту [14].

Ефективність роботи групи інспекторів залежить від ступеня їхнього знайомства з продуктом (чим більше, тим краще). З іншого боку,

іноді буває, що результати інспекції коду можуть створити також помилкову думку про продукт, якщо експерти почали інспекцію з коду, не розглядаючи вимог й опису проекту, хоча більшість помилок виявляється не лише на рівні проекту, а й на етапі визначення вимог та архітектури системи.

Отже, інспекція коду має справу з перевіркою вихідної програми на виявлення помилок у її записі, що не відображає справжньої її якості.

## 7.4. Класифікація відмов та помилок

За міжнародним стандартом ANSI/IEEE-729-83 робиться поділ усіх дефектів розробки програм.

**Помилка (error)** — це стан програми, при якому видаються неправильні результати, причиною котрих є недоліки (flaw) в операторах програми або в технологічному процесі її розробки, що призводить до неправильної інтепретації вхідної інформації, а отже, і до неправильного розв'язання.

**Дефект (fault)** у програмі є наслідком помилок розробника на кожному з етапів розробки і може утримуватися у вхідних або проектних специфікаціях, текстах кодів програм, в експлуатаційній документації тощо.

**Відмова (failure)** — це відхилення програми від функціонування або неможливість виконувати функції, визначені вимогами й обмеженнями, і розглядається як подія, що сприяє переходу програми в непрацездатний стан при помилках або відмовах у програмі або в середовищі функціонування.

Відмова може бути наслідком таких причин:

- помилкова специфікація або пропущена вимога, тобто специфікація точно не відображає того, які припущення робив користувач;
- специфікація може містити вимогу, яку неможливо виконати на даній апаратурі і програмному забезпеченні;
- проект програми може мати помилки (наприклад, базу даних спроектовано без захисту від несанкціонованого доступу користувача, а потрібен захист);
- проект програми може мати помилки (наприклад, опис компоненти містить алгоритм контролю несанкціонованого доступу, що не управляється коректно);
- програма може бути неправильною, тобто вона виконує невластивий алгоритм або його зроблено неповністю.

Таким чином, відмови, як правило, є результатами однієї або більше помилок у програмі, а також наявності всіляких дефектів.

#### 7.4.1. Помилки та причини появи їх на етапах ЖЦ

Є підходи до класифікації помилок у програмному забезпеченні відповідно до їхнього розподілу за етапами життєвого циклу і до джерел їхнього виникнення:

- ненавмисне відхилення розробників від робочих стандартів або планів реалізації;
- специфікації функціональних та інтерфейсних вимог виконано без дотримання стандартів розробки і под., що призводить до порушення функціонування програм;
- організація процесу розробки є недосконалою або має недостатнє керування з боку менеджера проекту ресурсами (людськими, технічними, програмними та іншими) і питаннями тестування й інтеграції елементів проекту.

Помилки виникають під час розробки програм на етапах життєвого циклу. Розглянемо їх нижче.

**Етап аналізу вимог.** У визначенні вхідної концепції системи та опису вхідних вимог замовника виникають помилки аналітиків, коли вони формують специфікації верхнього рівня і будують концептуальну модель предметної області.

Характерними помилками цього етапу є:

- неадекватність опису специфікаціями вимог кінцевих користувачів;
- некоректність специфікації взаємодії програмного забезпечення із середовищем функціонування або з користувачами;
- невідповідність вимог замовника до окремих і загальних властивостей програмного забезпечення;
- некоректність опису функціональних характеристик;
- незабезпеченість інструментальними засобами підтримки усіх аспектів реалізації вимог замовника тощо.

**Етап проектування компонент.** Помилки під час проектування компонент можуть виникати при описі алгоритмів, логіки керування, структур даних, інтерфейсів, логіки моделювання потоків даних, форматів введення — виведення тощо. В основі цих помилок лежать дефекти специфікації аналітиків та помилок проектувальників. До них належать помилки:

- визначення інтерфейсу користувача із середовищем;
- опису функцій (неадекватності формулювань у проекті мети та завдань окремих компонентів, що виявляються при перевірці проекту);
- визначення процесу опрацювання інформації або зв'язків між процесами (наслідок некоректного визначення взаємозв'язків компонентів та процесів);
- визначення даних і їхніх структур для окремих компонент та програмного забезпечення, що в цілому некоректно задані;

— опису алгоритмів модулів та їхньої логіки, що некоректно визначені в поданому проекті модуля;

- визначення умов виникнення можливих помилок у програмі;
- з причини порушення прийнятих для проекту стандартів та технологій.

**Етап кодування і налагодження.** На даному етапі виникають помилки, що є результатом дефектів проектування, помилок програмістів та менеджерів процесу розробки і налагодження. Причиною помилок є:

- безконтрольність допустимості значень вхідних параметрів, індексів масивів, параметрів циклів, вихідних результатів, ділення на 0 та ін.;
- неправильна обробка нерегулярних ситуацій під час аналізу кодів повернення від підпрограм, функцій, що викликались, тощо.
- порушення стандартів кодування (погані коментарі, нераціональне виділення модулів і компонентів тощо);
- використання одного імені для позначення кількох об'єктів або кількох імен на позначення одного об'єкта, незадовільна мнемоніка імен;
- неузгоджене внесення змін у програму кількома розробниками тощо.

**Етап тестування.** На цьому етапі помилки допускають тестувачі, а також програмісти, виконуючи технології збирання і тестування, вибору тестових наборів і сценаріїв тестування тощо. Відмови програмного забезпечення, викликані подібними помилками, мають виявлятися і не відбиватися на статистиці помилок та відмов компонентів і програмного забезпечення в цілому.

**Етап супроводження.** При супроводженні програмного забезпечення причиною помилок є дефекти експлуатаційної документації, слабкі показники модифікованості й зрозумілості програмного забезпечення, а також некомпетентність осіб, відповідальних за супровід та/або удосконалення програмного забезпечення. Залежно від сутності внесених змін на цьому етапі можуть виникати практично будь-які помилки, аналогічні раніше переліченим помилкам на попередніх етапах.

Усі помилки, що виникають у програмах, прийнято підрозділяти на класи [8]:

- логічні і функціональні помилки;
- помилки обчислень і часу виконання;
- помилки введення — виведення і маніпулювання даними;
- помилки інтерфейсів;
- помилки обсягу та інші.

Логічні помилки є причиною порушення логіки алгоритму, внутрішньої неузгодженості змінних в операторах, а також правил програмування. Функціональні помилки є наслідком неправильно ви-

значених функцій, порушення порядку застосування або неповноти реалізації їх тощо.

Помилки обчислень виникають через неточності вихідних даних і реалізованих формул, похибок методів, неправильного застосування операцій обчислень або операндів. Помилки часу виконання пов'язані з незабезпеченням необхідної швидкості обробки запитів або часу відновлення програми.

Помилки введення — виведення і маніпулювання даними є наслідком неякісної підготовки даних для виконання програми, збоїв під час занесення їх до бази даних або під час читання з неї.

Помилки інтерфейсу належать до помилок взаємозв'язку окремих елементів один з одним, що виявляється під час передачі даних між ними, а також при взаємодії із середовищем функціонування.

Помилки обсягу належать до даних і є наслідком того, що реалізовані методи доступу і розміри бази даних не задовольняють обсягів інформації й інтенсивності її обробки.

Наведені основні класи помилок властиві багатьом типам компонент програмного забезпечення і виявляються вони в програмах по-різному. Так, у роботі з базою даних виникають помилки представлення і маніпулювання даними, логічні помилки в завданні прикладних процедур обробки даних та ін. У програмах обчислювального характеру переважають помилки обчислень, а в програмах керування й обробки — логічні і функціональні помилки. У програмному забезпеченні з множини різнопланових програм реалізації кількох функцій можуть бути помилки кількох типів. Помилки інтерфейсів і порушення обсягу характерні для програмного забезпечення будь-якого типу.

Аналіз можливих типів помилок у програмах є необхідною умовою створення планів тестування і методів тестування для забезпечення правильності програмного забезпечення.

На сучасному етапі розвитку засобів підтримки розробки програмного забезпечення (CASE-технології, об'єктно орієнтовані методи та засоби проектування моделей і програм) передбачають таке проектування, при якому програмне забезпечення захищається від найбільш типових помилок і тим самим запобігається поява програмних дефектів.

#### 7.4.2. Зв'язок помилки з відмовою

Наявність помилки в програмі, як правило, призводить до відмови програмного забезпечення, коли воно функціонує. Для аналізу причинно-наслідкових зв'язків “помилка—відмова” у програмній компоненті є такі підходи:

- ідентифікація недоліків у технологіях проектування і програмування;

- взаємозв'язок недоліків процесу проектування і помилок, які допускає людина;

- класифікація відмов, недоліків і можливих помилок, а також дефектів на кожному етапі розробки програмного забезпечення;

- зіставлення помилок людини, які припускаються на визначеному етапі розробки, і дефектів в об'єкті як наслідку помилок специфікації проекту, моделей програм і под.;

- створення методології проектування, що передбачає перевірку і захист від помилок на всіх етапах життєвого циклу програмного забезпечення, а також виявлення дефектів на кожному етапі розробки;

- зіставлення дефектів та відмов програмного забезпечення для розробки системи їхніх взаємозв'язків і методика локалізації, збору й аналізу інформації про відмови і дефекти;

- розробка підходів до документування випробовувань програмного забезпечення.

Кінцева мета аналізу наслідкових зв'язків “помилка—відмова” полягає у визначенні методів та засобів тестування і виявлення помилок визначених класів, критеріїв завершення тестування на множині наборів даних, а також визначення шляхів удосконалення організації процесу розробки, тестування, супроводу програмного забезпечення.

Наведемо таку класифікацію відмов:

- апаратні, при яких загальносистемне програмне забезпечення не працездатне;

- інформаційні, викликані помилками у вхідних даних та інформаційних структурах, у тому числі при передачі даних каналами зв'язку, в збоях пристроїв введення (наслідок апаратних відмов);

- ергатичні, викликані помилками оператора в його взаємодії з машиною (ця відмова є вторинною відмовою і може призвести до інформаційної або функціональної відмов);

- програмні, при наявності помилок та інші.

Деякі помилки можуть бути наслідком недоробок під час визначення вимог, проекту, генерації вихідного коду або документації, а з іншого боку, вони породжуються в процесі розробки програми або під час розробки інтерфейсів окремих елементів програми (порушення порядку параметрів, менше або більше параметрів тощо). На рис. 7.3 показано подібні випадки помилкових ситуацій і причини їхньої появи в кожній розробці.

#### 7.4.3. Джерела помилок

Помилки можуть виникати під час розробки проекту, компонент, коду, документації, тобто виявлятися в кількох місцях розробки або супроводу.

Деякі помилки в програмі можуть бути наслідком недоробок під час визначення вимог, проекту, генерації коду або документації. З іншого боку, помилки породжуються в процесі розробки програми



Рис. 7.3. Види помилок на етапах життєвого циклу системи

або інтерфейсів її елементів (наприклад, при порушенні порядку задання параметрів зв'язку — менше або більше, ніж потрібно, тощо).

Причиною появи помилок часто є незрозумілість вимог замовника і неточність їхньої специфікації в документах проекту. Це призводить до того, що реалізуються деякі функції системи, які будуть працювати не так, як вимагає замовник. У зв'язку з цим виробляються домовленості замовника з розробником для усунення непорозуміння й уточнення вимог до системи.

Команда розробників системи може також допустити помилку в синтаксисі і семантиці опису системи. Компілятор відшукує подібні помилки до початку роботи програми. Проте деякі помилки може бути не виявлено. Наприклад, якщо синтаксис операторів правильний, а деякі індекси або значення змінних цих операторів — хибні.

Правильно відтрансльована програма може також видавати помилки, якщо для її виконання неправильно задано деякі граничні умови. Аналогічна ситуація виникає, якщо використовується повторна компонента, значення параметрів звернення до якої задано неправильно.

Виходячи з того, що кожна організація з розробки програмного забезпечення (особливо загальносистемного призначення) стикається з проблемами тестування для пошуку помилок, вона змушена класифікувати типи помилок і визначати своє ставлення до цих помилок. На основі багаторічної діяльності в галузі створення програмного забезпечення багато фірм пропонували свою класифікацію помилок, характеризуючи їхню появу під кількома кутами зору: процес розробки, функції проектованої системи, типові ділянки і дії. Відомо чимало підходів до класифікації помилок, розглянемо деякі з них.

Фірма "IBM" розробила підхід до класифікації помилок, названий ортогональною класифікацією дефектів [14]. Підхід передбачає розбивку помилок за категоріями із ступенем відповідальності за них розробників.

Схема класифікації є продукто- і організаційно незалежна і може застосовуватися на всіх стадіях розробки програмного забезпечення різноманітного призначення. У табл. 7.2 дається список помилок відповідно до даної класифікації. У ній розробнику надається можливість ідентифікувати не лише типи помилок, а й місця, в яких пропущено або зроблено помилки. Пропущена помилка, як правило, засвідчує, що немає результату, якщо в програмі є неініціалізована змінна або ініціалізованій змінній надано неправильне значення.

Таблиця 7.2. Ортогональна класифікація дефектів, виконана IBM

Контекст помилки	Класифікація дефектів
Функція	Помилки інтерфейсів кінцевих користувачів програмного забезпечення, що викликаються апаратурою або пов'язані з глобальними структурами даних
Інтерфейс	Помилки у взаємодії з іншими компонентами, у викликах, макросах, керуючих блоках або в списку параметрів
Логіка	Помилки в програмній логіці, не охопленій валідацією, а також у використанні значень змінних
Присвоювання	Помилки в структурі даних або в ініціалізації змінних окремих частин програми
Зациклювання	Помилки, викликані ресурсом часу, реальним часом або поділом часу
Середовище	Помилки в репозиторії, в керуванні змінами або в контрольованих версіях проекту
Алгоритм	Помилки, пов'язані із забезпеченням ефективності, коректності алгоритмів або структур даних системи
Документація	Помилки в записях документів супроводження або в публікаціях

Ортогональність схеми класифікації полягає в тому, що будь-який її термін належить до точно однієї категорії. Іншими словами, простежувана помилка в системі має бути в одному з класів, що дає можливість двом розробникам класифікувати помилку однаковим чином.

Градї Буч (1997 р.) [14] описав підхід до класифікації помилок, ввівши кілька категорій знайдених помилок. На рис. 7.4 показано схему і модель класифікації помилок за трьома вимірами:

- місце перебування помилки;
- тип помилки;
- характер (пропуск, неясності, неправильності, обмін тощо);

При цьому SW позначає програмне забезпечення, а HW — апаратуру.

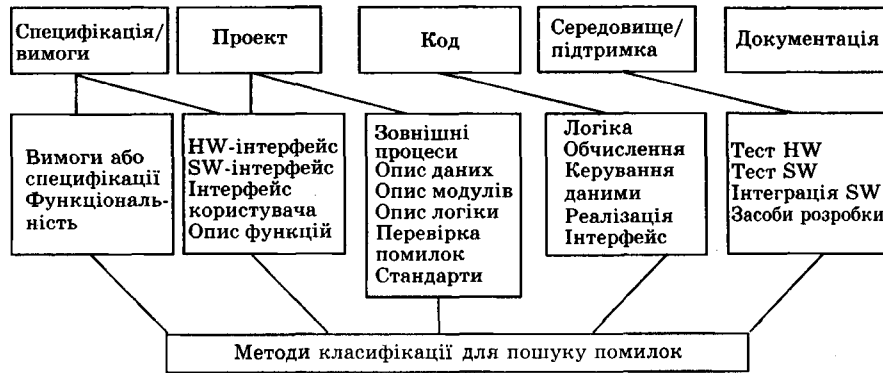


Рис. 7.4. Класифікація помилок за Бучем

Фірма "Hewlett-Packard" застосовувала класифікацію Буча, після чого встановила процентне співвідношення помилок, що виявляються в програмному забезпеченні на стадіях розробки (рис. 7.5) [14].

У керуванні даними

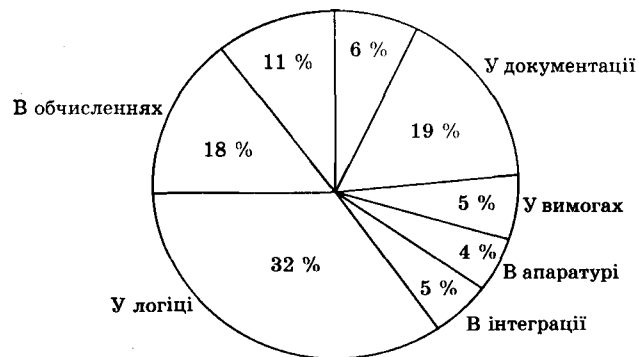


Рис. 7.5. Процентне співвідношення помилок при розробці програмного забезпечення

## 7.5. Тести програм і систем

Для перевірки правильності програм спеціально розробляються тести і тестові дані. Під *тестом* розуміють певну програму, призначену для перевірки працездатності іншої програми і виявлення в ній помилкових ситуацій. Тестову перевірку можна провести також шляхом введення в програму, що перевіряється, додаткових операторів, які будуть сигналізувати про перебіг виконання її й одержання результатів.

Тестові дані слугують для перевірки роботи системи і готуються по-різному: генератором тестових даних, проектною групою на основі документів або файлів, користувачем із специфікації вимог тощо. Дуже часто розробляються спеціальні форми вхідних документів, в яких відображається процес виконання програми за допомогою тестових даних.

### 7.5.1. Види тестів

Є такі види тестів перевірки:

- повнота функцій;
- узгодженість інтерфейсів;
- структура програми;
- обчислення і коректність виконання функцій;
- правильність функціонування в заданих умовах;
- надійність виконання програм;
- ефективність захисту від збоїв апаратури і невиявлених помилок;
- зручність застосування, супроводження тощо.

Тестові дані готуються як для перевірки окремих програмних елементів, так і для груп програм або комплексів на стадіях процесу розробки. На рис. 7.6 наведено класифікацію тестів перевірки за об'єктами тестування на основних стадіях розробки.

Багато типів тестів готує сам замовник для перевірки роботи програмної системи. Структура й зміст тестів залежать від виду тестованого елементу, яким може бути: модуль, компонента, група компонент, підсистема або система. Деякі тести залежать від мети й необхідності знати: чи працює система відповідно до її проекту, чи задоволено вимоги і чи бере участь замовник у перевірці роботи тестів. Для проведення тестування створюється спеціальна команда (група) тестувачів.

Залежно від завдань, що ставляться перед тестуванням програм, ця команда складає тести перевірки проміжних результатів проектування елементів системи на стадіях життєвого циклу, а також створює тести іспитів остаточного коду системи.

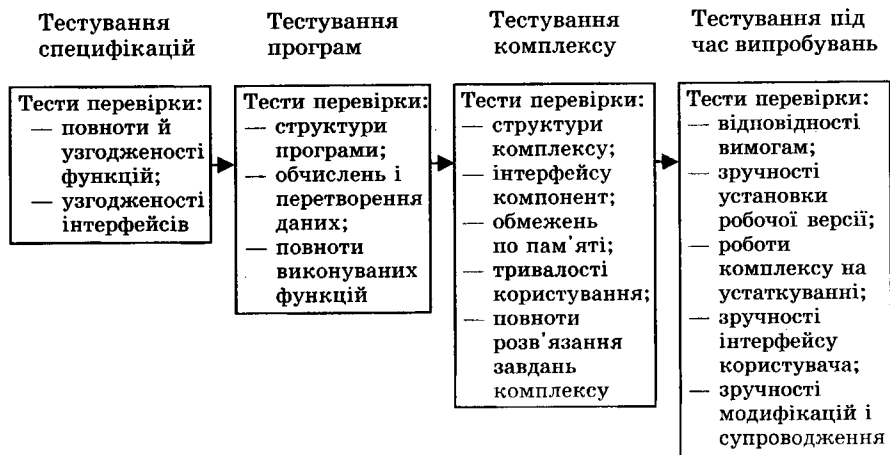


Рис. 7.6. Класифікація тестів перевірки

**Тести інтегрованої системи.** Тести для перевірки окремих елементів системи і тести інтегрованої системи мають загальні й відмінні риси.

Так, на рис. 7.7 як приклад наведено кроки стадії інтеграції системи з готових після тестування елементів, задано зв'язки між кроками тестування, які інтегруються програмною системою.

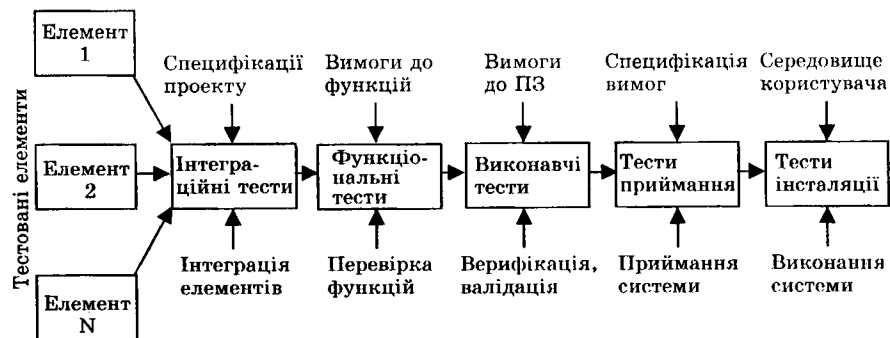


Рис. 7.7. Тести і стадії інтеграційного тестування

Для кожного кроку команда тестувачів готує тести і набори даних, що використовуються для перевірки станів інтегрованої системи і відповідності прийнятим обмеженням на кожному кроці та вимогам до елемента тестування.

Перший крок цього процесу — інтеграція елементів, що ґрунтується на інтеграційних тестах, утворюваних за специфікаціями проекту. Розглянемо цей крок докладніше на прикладі схеми (рис. 7.8) інтеграції окремих елементів.

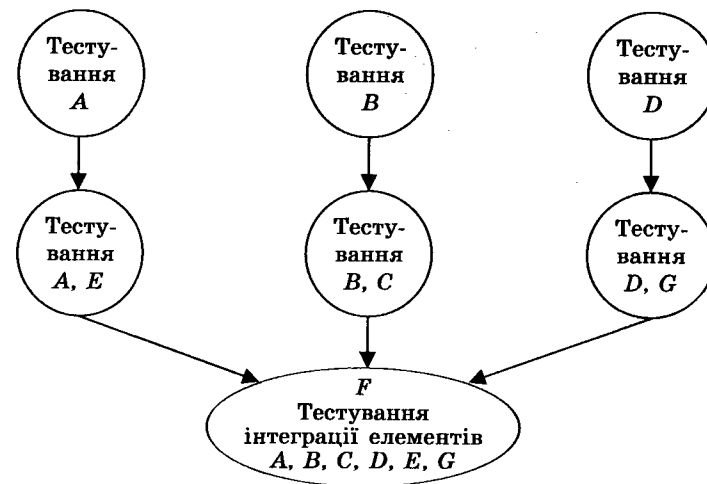


Рис. 7.8. Інтеграційне тестування компонент

Кожна компонента цієї схеми тестується окремо від інших компонент за допомогою тестів, які включають набори даних і сценарії, складені відповідно до їхніх типів і функцій, специфікованих у проекті системи. Тестування проводиться в контрольному середовищі на визначеній множині тестових даних та операціях, виконуваних над ними.

Тести забезпечують перевірку внутрішньої структури, логіки і граничних умов виконання кожної компоненти.

Відповідно до наведеної схеми, спочатку тестуються компоненти A, B, D незалежно одна від одної і кожна з окремим тестом. Після перевірки їх виконується такий етап, як перевірка інтерфейсів для їхньої наступної інтеграції, суть якої полягає в аналізі виконання операторів виклику  $A \rightarrow E$ ,  $B \rightarrow C$ ,  $D \rightarrow G$  на нижчих рівнях графу: компоненти E, C, G. При цьому передбачається, що зазначені компоненти, які викликаються, так само має бути налагоджено окремо. Аналогічно перевіряються всі звертання до компоненти F, що є ланкою, яка пов'язує елементи.

При цьому можуть виникати помилки в разі неправильного задання параметрів в операторах виклику або коли обчислюються процедури чи функції. Знайдені помилки у зв'язках усуваються, а потім повторно перевіряється зв'язок з компонентою F у вигляді "трийок": компонента — інтерфейс — компонента.

Наступним кроком тестування системи (рис. 7.8) є перевірка функціонування системи за допомогою функціональних тестів, котрі включають сценарії перевірки функцій і вимог до них. В основі цього тестування лежать проектні специфікації та функціональні вимоги до системи.

Після перевірки системи на функціональних тестах йде перевірка системи на виконавчих і випробувальних тестах, основаних на вимогах до програмного забезпечення, до апаратури і виконуваних функцій. Випробувальному тесту передують верифікація і валідація програмного забезпечення.

Тест для перевірки роботи всієї системи є складнішим, до нього включаються набори даних для перевірки функцій на відповідність їхніх описів та вимог до них. Тест випробувань системи відповідно до вимог замовника перевіряється в реальному середовищі, в якому система буде використовуватися.

Тест інсталяції системи слугує для перевірки виконання всієї системи в середовищі користувача.

### 7.5.2. Команда тестувачів

За функціональні і виконавчі тести відповідає розробник, а замовник більше впливає на складання випробувальних та інсталяційних тестів.

Як правило, команда тестувачів не залежить від штату розробників програмних систем. Деякі члени цієї команди є досвідченими тестувачами або навіть професіоналами в цій галузі. До них належать аналітики, програмісти, інженери-тестувачі, котрі присвячують увесь час проблемам тестування систем. Вони мають справу не лише зі специфікаціями, а й з методами та засобами тестування, організують створення і виконання тестів на машині. Тестувачів включають у процес розробки із самого початку створення проекту для складання тестових наборів та сценаріїв, а також графіків виконання тестів.

Фахові тестувачі працюють разом з групою управління конфігурацією, щоб забезпечити їх документацією та іншими механізмами для узгодженості тестів між собою і з вимогами проекту, конфігурацією та кодом. Вони розробляють методи і процедури тестування. До цієї команди включаються додаткові люди, знайомі з вимогами системи або з підходами до її розробки. Аналітиків включають як членів команди, тому що вони розуміють проблеми визначення специфікацій замовників.

Багато спеціалістів порівнюють тестування системи із створенням нової системи, в якій аналітики відображають потреби та мету замовника, працюючи разом з проектувальниками і домагаючись з'ясування ідей та принципів роботи системи, щоб виконати поставлені завдання тестування.

Проектувальники системи додають команді тестувачів проектні завдання, тому що їм відомі принципи декомпозиції системи на підсистеми і функції, а також принципи їхньої роботи. Після проектування тестів і тестових покриттів команда тестувачів підключає проєктантів для аналізу можливостей системи.

Тести і тестові сценарії є прямим відзеркаленням вимог та проекту в цілому. Перспективи управління конфігурацією системи визначаються саме цією командою. Помилки, які ще трапляються в програмі, і зміни в системі відображаються в документації, вимогах, проєкті, а також в описах вхідних та вихідних даних або в інших розроблюваних артефактах. Внесені зміни в процесі розробки приводять до модифікації тестових сценаріїв або в більшій частині до зміни планів тестування. Фахівці з управління конфігурацією враховують ці зміни і координують упорядкування тестів.

До команди тестувачів входять також користувачі. Вони оцінюють отримані результати, зручність використання і людський чинник, а також висловлюють свою думку про принципи роботи системи на початкових етапах проєкту.

Уповноважені замовника планують роботи для тих, хто буде використовувати і супроводжувати систему. При цьому вони можуть привнести деякі зміни в проєкт, викликані неповнотою заданих вимог, і сформулювати системні вимоги для проведення верифікації системи й ухвалення рішення про її готовність та корисність.

### 7.5.3. План тестування

Для проведення тестування створюється план (Test Plan), в якому описуються стратегії, ресурси і графік тестування окремих компонент та системи в цілому. У проведенні тестування беруть участь члени команди, виконуючи визначені ролі в цьому процесі. План стратегії включає визначення місця тесту в кожному процесі, ступінь покриття програми тестами і відсоток тестів, що виконуються із спеціальними результатами. Тестові інженери створюють множину тестових сценаріїв (Test Cases), кожен з яких перевіряє результат взаємодії між актором і системою на основі визначених перед- та постумов використання таких сценаріїв. Сценарії належать здебільшого до тестування за типом “біла скринька” і орієнтовані на перевірку структури та операції інтеграції компонентів системи.

Для проведення тестування тестові інженери пропонують процедури тестування (Test Procedures), що включають валідацію об'єктів та верифікацію тестових сценаріїв відповідно до план-графіка. Оцінка тестів (Test Evaluation) полягає в оцінці результатів тестування та ступеня покриття програм сценаріями і статусу отриманих помилок. На рис. 7.9 наведено коло обов'язків тестового інженера.

Тестувач інтеграції системи проводить тестування інтерфейсів й оцінку результатів виконання відповідних тестів. І, нарешті, тестувач системи є відповідальним за виконання системних тестів і тестування, проведеного попередніми членами команди. При виконанні сис-

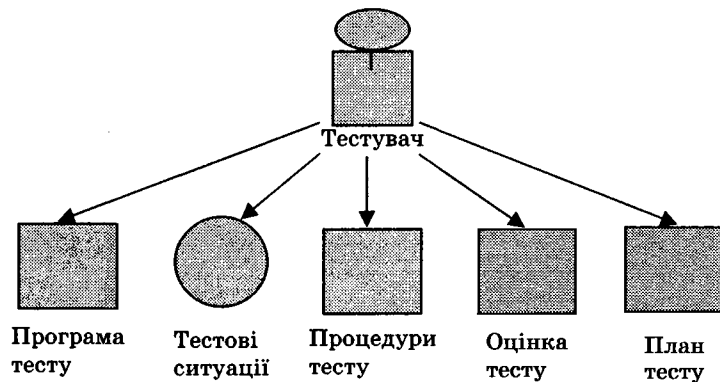


Рис. 7.9. Відповідальності інженера тестів

темних тестів, як правило, знаходяться дефекти, що є результатом глибоко прихованих похибок у програмах, які виявляються під час тривалої прогонки системи на тестових даних і сценаріях.

## 7.6. Засоби автоматизації процесу тестування

До цього часу розроблено багато різноманітних засобів пошуку помилок, дефектів та відмов у програмах. За призначенням ці засоби поділяються відповідно до типів тестування (статичного, динамічного тощо) програмного забезпечення. Як правило, всі засоби об'єднуються базою даних проектування системи. Вона містить усі компоненти, налагоджувальні контрольні дані і тести, а також результати тестування й інформацію про документування системи та процесу тестування.

База проекту підтримується спеціальними інструментальними засобами (типу CASE [20,21]). Наприклад, засобом ведення аналізу предметних областей з відповідним набором даних про їхні об'єкти, відношення між ними, потоки даних та їхній зв'язок з об'єктами тощо. База проекту зберігає також початкові й еталонні дані, які використовуються для зіставлення даних, накопичених у базі з даними, які отримані під час тестування.

Далі подається характеристика основних засобів тестування.

**Засоби статичного тестування.** До них належать засоби розрахунку тривалостей виконання модулів та їхніх характеристик. Вони дозволяють отримувати середні значення і розподіли розрахунків тривалостей аналітично, без виконання програми на машині. Внаслідок розрахунків виявляються компоненти програми, які потребують багато часу для виконання і перевірки виконаності програмного забезпечення в реальному часі. Ці дані дозволяють знайти деякі помил-

ки порушення надійності функціонування через невідповідність тривалості виконання програмного забезпечення потребам реального часу.

**Засоби динамічного налагодження та тестування.** Їх можна розподілити на два типи. Перший безпосередньо забезпечує виконання програм відповідно до тестових завдань, другий тип — допоміжні засоби, які обчислюють результати виконаного тестування і проводять необхідні коригування програм. До другого типу належать засоби трансляції завдань з мови налагодження, виконання програм з налагодженого завдання і реєстрації даних про результати тестування. Тестові значення перетворюються у форму, придатну для виконання налагодженої програми. Оператори налагодженого завдання об'єднуються з програмою, що тестується, або готуються для виконання в режимі інтерпретації.

**Засоби управління виконанням.** Ці засоби реалізують виконання виготовленої програми з налагодженого завдання. Під час обробки завдань і тестів провадиться вибір результатів відповідно до завдання.

Під час обробки тестів проводиться селекція результатів тестування відповідно до операторів налагодженого завдання, а також зіставлення їх з еталонними значеннями. Деякі результати зберігаються для дальшого використання їх.

У процесі виконання програм у режимі тестування результати відображаються на дисплеї, наприклад у графічній формі (шляхи проходження по графу програми), у вигляді послідовності діаграм UML, а також інформації про відмови та помилки або конкретні значення вихідних параметрів програми. Ці дані аналізуються розробниками для формулювання висновків про напрями подальшої перевірки правильності програм або завершення їх.

**Засоби планування.** Засоби автоматизації планування тестування призначені для підготовки інформації трьох видів:

- про цикли;
- про шляхи виконання програми;
- про предикати, котрі керують виконанням шляхів, і про межі областей змін параметрів та змінних.

Такі засоби спочатку будують стратегію і шляхи тестування відповідно до умов у предикатах, виключаючи виконання циклів. Потім у діалозі запрошуються реальні значення ймовірного розгалуження у вершинах графу та характеристик ітерації циклів. Цикли, як правило, перевіряються окремо з використанням вхідних даних про ітерації. Після перевірки вони вилучаються зі шляхів виконання програми. Під час планування шляхів виконання готуються тести, критерії і вхідні значення.

Усі перелічені дані використовуються засобами автоматизації при багаторазовому проходженні тестів за виконуваною програмою.

**Засоби документування результатів тестування.** Відповідно до чинного стандарту ANSI/IEEE 829, результати налагодження і тестування програмних систем на всіх етапах життєвого циклу треба збирати в спеціальних документах, які створюються автоматично і містять:

- загальний опис завдань, призначення та зміст програмної системи, а також опис її функцій відповідно до вимог замовника;
- опис технології розробки системи;
- опис планів тестування різних об'єктів, необхідних ресурсів, відповідних спеціалістів для проведення тестування та технологічних засобів;
- специфікацію тестів, контрольних прикладів, критеріїв та обмежень оцінки результатів програмного продукту, а також процесу тестування;
- облік тестування, звіт про аномальні події, відмови та дефекти з підсумковими результатами тестування компонентів і системи в цілому.

Отже, у тестуванні використовується багато спеціальних засобів, які дають змогу поліпшити показники створюваної системи. Крім того, є окремі засоби, орієнтовані на тестування таких складних систем, як критичні системи та системи реального часу тощо.

## Контрольні запитання і завдання

1. У чому полягає зміст процесу тестування?
2. Назвіть методи тестування.
3. Поясніть значення термінів “чорна скринька”, “біла скринька”.
4. Назвіть основні методи доведення коректності програм і базис цих методів.
5. Визначте типи логічних операцій, використовуваних при логічному доведенні коректності програм.
6. У чому полягає відмінність техніки формального доведення від символічного виконання програм?
7. Сформулюйте основні завдання верифікації та валідації програм.
8. Назвіть об'єкти тестування і підходи до тестування їх.
9. Яка є класифікація типів помилок у програмах?
10. Визначте основні етапи життєвого циклу тестування програмного забезпечення.
11. Наведіть класифікацію тестів для перевірки програмного забезпечення.
12. Які завдання виконує група тестування?

## Література до глави 7

1. *Лунаев В.В.* Отладка сложных программ — М.: Энергоатомиздат, 1993. — 296 с.
2. *Myers G.J.* The Art of Software Testing. — New York: Wiley, 1979. — 413 p.
3. *Wang Y., King J., Kourt J., Ross M., Staples S.* On testable object-oriented programming // Software Engineering Notes. — 1997. — V. 22 — № 4. — P. 84—90.
4. *Perry D.E., Kaiser C.E.* Adequate testing and object-oriented programming // Journal of Object-Oriented Programming. — 1990. — January/February. — P. 13—19.
5. ANSI / IEEE Std. 10122—1986. Standard for Software Verification and Validation Plans // IEEE. — New York, 1986. — 61 p.
6. *Dolores R. Wallase M. Ippolito, b. Cuthill.* Reference Information for the Software Verification and Validation Process // NIST Special Publication, 1996. — 500—234. — 80 p.
7. *Herhart S.L.* Program Verification in the 90's. // Proc.Conf. on Computing in the 1980's, 1978. — P.80—89.
8. *Grossman D., McCabe C.* Performance Testing a Large Finance Application // IEEE Software. — 1996. — Sept. — P. 50—60.
9. *Андон Ф.И., Лаврищева Е.М.* Методы инженерии распределенных компьютерных систем. — К.: Наук. думка, 1997. — 229 с.
10. *Clint M., Hoare C.A.R.* Program proving: jumps and functions // Acta Informatica. — 1972. — 1. — № 3. — P. 214—224.
11. *Burstall R.M.* Program proving as hand simulation with a little induction. — Proc. IFIP Congress 74, North-Holland, 1974. — P. 80—89.
12. IEEE Software. Measurement. — March/April, 1997.
13. *Jacobson J.* Object-oriented Software Engineering. — Revised Printing. — Addison-Wesley, 1995. — 528 p.
14. *Pfleeger S.L.* Software Engineering. Theory and Practice. — Prentice Hall, 1998. — 576 p.
15. *Dijkstra T.W.* Finding the Correctness proof of a concurrent program // Proc.Conf. Nederland Acad.Wetenach. — 1978. — 81. — № 2. — P. 207—215.
16. CASE-93. Proceeding Sixth Intern./Workshop on Computer Aided Software Engineering. — Singapore. — 1993. — July 19—23. — 418 p.
17. *Бабенко Л.П.* Проблемы повторного использования в программной инженерии // Кибернетика и системный анализ. — 1999. — № 2. — С. 155—166.
18. *Тейер Т., Липов М., Нельсон Э.* Надежность программного обеспечения. — М.: Мир, 1981. — 325 с.
19. *Коротун Т.М.* Совершенствование процесса тестирования программного обеспечения // Проблемы программирования. — 1998. — № 3. — С. 59—64.

## Глава 8. СУПРОВОДЖЕННЯ ПРОГРАМНИХ СИСТЕМ

Терміном *супроводження* будемо позначати будь-які роботи з внесення змін до системи після того, як її було передано користувачеві для експлуатації. На відміну від обладнання, яке з часом потребує ремонту або профілактичного запобігання пошкоджень, програмне забезпечення не “зношується” ніколи, тому процес супроводження націлений на підтримку передовсім еволюціонування системи, тобто на зміну її функцій та властивостей. Серед причин, котрі можуть зумовити потребу змін, можна назвати типові, а саме:

- виявлення дефектів функціонування системи під час експлуатації, не виявлених на етапі тестування (зміни, за які несе відповідальність розробник);
- з'ясування під час експлуатації, зробленої замовником, що він недостатньо або неповно висловив свої вимоги, завдяки чому система не відповідає окремим потребам замовника (зміни, за які несе відповідальність постановник проблеми — так званий аналіст);
- зміна умов діяльності замовника, які тепер не відповідають раніше поставленим вимогам (наприклад, змінилися податкове законодавство або місцева регуляція правил бізнесу, змінилися способи комунікації замовника з бізнес-партнерами або відбувся перерозподіл їхніх ролей у бізнесі тощо). Такі зміни важко передбачити кожній із сторін, котра брала участь у формулюванні вимог, оскільки вони викликані об'єктивною реальністю життя.

Висновком з усіх перелічених випадків, без урахування того, “чия провина викликала зміни”, може бути ще раз повторено тезу:

“Всяку передану замовникові програмну систему має бути змінено або виведено з використання”.

При цьому кожна внесена зміна сприяє зростанню складності системи і заважає можливості зрозуміти її сутність, якщо не вжити спеціальних запобіжних заходів. Мається на увазі, що локальні зміни часто реалізуються на рівні кодування шляхом “латання” основного тексту. Якщо ж при цьому не відобразити внесені зміни в моделі вимог, то з часом розібратися в таких програмах неможливо навіть їхнім авторам.

Саме тому, як засвідчують експерти, процес внесення змін є достатньо дорогим — оцінки його вартості сягають від 60 % до 80 % від загальної вартості розробки.

Серед видів діяльності із здійснення супроводження прийнято розрізняти такі:

- коригуюче супроводження. Це внесення коректив для усунення похибок, які було знайдено після передачі системи до експлуатації. За оцінками експертів, коригуюче супроводження становить 25 % зусиль супроводження в цілому;
- адаптивне супроводження. Це адаптація продукту до змінених обставин використання після передачі системи в експлуатацію. За оцінками експертів, адаптивне супроводження становить 25 % зусиль супроводження в цілому. Адаптивне супроводження може бути суттєво спрощено, якщо вдасться прогнозувати змінність (варіантність) певних властивостей і передбачити спектр можливих змін та відповідних механізмів адаптації до них;
- попереджувальне супроводження. Це діяльність із забезпечення адаптивного супроводження на старті розробки, як зазначено вище.

У главі 12 обговорюються шляхи визначення властивостей проекту, для яких доцільно зафіксувати варіантність, представлення виявленої варіантності на моделях аналізу вимог та проектування. Обговорюються також типові механізми адаптації згаданих моделей до потрібних змін, тобто конкретизації варіантності, яка найчастіше може здійснюватися як вибір однієї з визначених наперед можливостей або як довизначення користувачем допоміжних послуг, потрібних для конкретного застосування;

- вдосконалювальне супроводження. Це вдосконалення продукту відповідно до нових вимог після передачі системи в експлуатацію. За оцінками експертів, вдосконалювальне супроводження становить 50 % зусиль супроводження в цілому;
- запобіжне супроводження. Це запобігання дефектам та пошкодам використання після передачі системи в експлуатацію. За оцінками експертів, запобіжне супроводження становить 4 % зусиль супроводження в цілому.

**Проблеми супроводження.** Найперша з проблем супроводження — це проблема персоналу. Внесення змін можливе тільки за умови розуміння сенсу того продукту, який потребує змін, та в чому полягають зміни. Зазначимо, що ключ до послаблення проблеми розуміння програм треба шукати в нотаціях представлення продуктів, які виробляються внаслідок діяльності спеціалістів на кожному з етапів життєвого циклу розробки програмних систем. Представлені у главах 3—5 моделі аналізу вимог та розробки якраз орієнтовані на досягнення порозуміння між особами-в-ролі, котрі беруть участь у створенні програмної системи.

Тож другою проблемою супроводження можна вважати використання формалізованих моделей документування на всіх стадіях життєвого циклу розробки (аналізу вимог, проектування, тестування тощо).

Важливою передумовою спрощення процесу супроводження є застосування спіральної моделі життєвого циклу розробки (див. главу 2.). Потреба у внесенні змін може з'ясуватися на будь-якій стадії життєвого циклу, і зміни може бути внесено також на будь-якій стадії — наприклад, шляхом внесення деяких додаткових фрагментів, які коригують попередню версію тексту коду (такий шлях на жаргоні програмістів називають “латанням” програми, а внесені фрагменти — “латками”). Якщо не відобразити внесені “латки” в продуктах усіх попередніх стадій шляхом внесення відповідних коректив до моделі аналізу вимог, моделі проекту тощо, то вони перестануть бути моделями “латаної” реалізованої системи. Тоді для внесення наступних змін доведеться вже розбирати її вихідні коди, що для великої системи надто дорого, а частіше неможливо.

Тож третя проблема супроводження — при внесенні змін адекватно коригувати всі використовувані моделі на всіх рівнях прийняття проектних рішень.

Для втілення спіральної моделі життєвого циклу розробки необхідним є *трасування*, тобто встановлення відповідності між локалізацією кожного з елементів моделі продукту будь-якої стадії і його відображенням у моделі іншої стадії. Наприклад, трасування передбачає встановлення асоціацій між класами об'єктів аналізу вимог та класами мови програмування C++, якщо другі є реалізацією перших. Так само відношення трасування може поєднувати моделі поведінки об'єктів та модулі вихідними мовами програмування, котрі є моделями реалізації зазначеної вище поведінки.

Звідси випливає четверта проблема супроводження — забезпечення трасування вимог для всіх моделей подальших етапів життєвого циклу розробки включно з інструкціями та навчальними засобами для кінцевого користувача. Вони дозволять уникнути недостатнього розуміння користувачем правил використання системи через недоліки документації.

П'ята проблема має моральний відтінок: супроводження вимагає зусиль виконавців такої самої високої професійної підготовки, як і нові розробки, але престиж діяльності супроводження значно нижчий. Та й виконавців нова розробка завжди приваблює більше, ніж усілякі доопрацювання. Тому корисною рекомендацією може бути ротація ролей учасників розробок.

Шоста проблема є проблемою тестування. По-перше, внесення змін потребує перетестування системи, бо зміни можуть порушити правильність роботи системи. По-друге, перетестування діючої системи вимагає її призупинення, що для систем керування реальними

об'єктами може виявитись небажаним, навіть неможливим. У цьому разі доводиться створювати дублікат або спеціальний макет системи.

Сьома проблема — це вивчення так званого побічного ефекту внесених змін, який полягає у виникненні ризику їхнього небажаного впливу на процеси та об'єкти, міняти які не було наміру. Для запобігання побічному ефекту треба провести спеціальну експертизу на виявлення впливу потрібних змін, по-перше, на представлення моделей для різних етапів життєвого циклу розробки (так зване горизонтальне трасування змін), а по-друге, вплив зміни одного з елементів певної моделі на інші елементи тієї самої моделі або елементи інших моделей, відповідних тій самій стадії життєвого циклу (наприклад, вплив зміни атрибутів одного класу на операції інших класів або на модель сценаріїв). Інколи для систем великого розміру доводиться навіть застосовувати спеціальні інструментальні засоби, наприклад, генератори взаємних посилань тощо.

В процесі супроводження можливе виникнення конфліктів між різними пріоритетами менеджменту: вносити зміни шляхом швидкого “латання” чи елегантного реструктурування системи. Друге сприятиме властивості системи бути зрозумілою, а можливо, й ефективнішою.

Плануючи розробку, можна передбачити ще на її старті запобіжні засоби для спрощення (а тим самим і здешевлення) супроводження. Серед них найдійовішими є планування передбаченої варіантності (див. главу 12.4) та засоби керування конфігурацією (див. главу 10).

## Контрольні запитання і завдання

1. Визначте мету і завдання супроводження програмного забезпечення.
2. Які види діяльності розрізняють при виконанні супроводження?
3. Дайте стислу характеристику проблем, що виникають при супроводженні.

## Глава 9. АНАЛІЗ І ДОСЯГНЕННЯ ЯКОСТІ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Терміном *якість програмного забезпечення* позначаємо сукупність властивостей, що визначають спроможність забезпечення задовольнити запити замовника, які він висловив як вимоги до розробки.

Аналіз якості у програмній інженерії орієнтований на:

- досягнення необхідної якості програмного забезпечення відповідно до встановлених критеріїв;
- верифікацію і валідацію (verification and validation — V&V) на етапах життєвого циклу та оцінку якості виробленого програмного продукту (див. главу 7);
- забезпечення надійності як основної характеристики гарантії якості програмного забезпечення (SQAs — Software Quality Assurance [1,2]).

Ці напрями програмної інженерії розглядаються на всіх етапах життєвого циклу, тобто аналіз і забезпечення якості проводиться за всіма видами діяльності у вирішенні завдань планування, розробки і підтримки процесів створення програмного забезпечення.

Створення якісного продукту сприяє задоволенню вимог замовника. Програмне забезпечення без визначеного рівня якості є індикатором помилок. Для досягнення якості здійснюються пошук та анулювання помилок на всіх етапах інженерної діяльності з розробки програмного забезпечення. Цей процес визначається ступенем підготовки й організації спільної діяльності людей, а також адекватністю стандартів, методології та засобів підтримки процесів розробки програмного забезпечення.

Процес специфікації вимог може породити хиби в проекті, код програми може не задовольняти вимог і мати помилки, яких не виявлено на етапах життєвого циклу, що призводить до великих проблем, у тому числі і до катастрофічних. Інженер програмного забезпечення, щоб було створено якісний продукт, має користуватися різноманітними технологіями і стандартами побудови якісного програмного забезпечення.

Для проведення аналізу якості, а також верифікації і валідації програмного забезпечення потрібні відповідні знання про організацію управління якістю, про процеси аналізу якості, метою яких є забезпечення додаткових гарантій досягнення заданої якості та її поліпшення.

Аналіз якості включає початкову оцінку, яка відповідає процесам виробництва якісного програмного забезпечення: моніторингу, плануванню, виконанню, зміні або підтримці.

Досягнення якості, як основного атрибута програмного забезпечення широкого класу систем, залежить від процесу проектування (див. главу 4), який має бути планованим і включати систематизований набір дій із забезпечення адекватності й довіри до продукту, створеного відповідно до технічних вимог. При цьому верифікація і валідація програмного забезпечення належать до управління якістю і є важливими процесами забезпечення якості продукту на етапах життєвого циклу.

Таким чином, аналіз якості є діяльністю, що включає процеси управління, інфраструктуру програмної інженерії, тестування, інженерію вимог.

### 9.1. Визначення якості програмного забезпечення

Основні стандартні положення зі створення якісного продукту й оцінки рівня досягнутої якості зафіксовано в міжнародних [1—3] та вітчизняних стандартах [4—6]. Згідно з цими стандартами виділяються два процеси забезпечення якості протягом життєвого циклу програмного забезпечення:

- гарантія (підтвердження) якості програмного забезпечення, що є результатом певних дій на кожній стадії життєвого циклу з перевірки й підтвердження відповідності програмного забезпечення стандартам та процедурам, орієнтованим на досягнення якості;
- інженерія якості як процес надання продуктам програмного забезпечення властивостей надійності, супроводження й інших характеристик якості.

Ці процеси досягнення якості призначено:

а) для керування, розробки і забезпечення гарантій відповідно до зазначених стандартів та процедур;

б) для керування конфігурацією (ідентифікація, урахування стану і дії з аутентифікації), ризиком та проектом відповідно до стандартів і процедур;

в) для контролю базової версії програмного забезпечення тощо.

Виконання перелічених процесів включають такі дії:

— оцінка стандартів і процедур, що виконуються при розробці програм;

— ревізія керування, розробки і забезпечення гарантії якості програмного забезпечення, а також всієї проектною документації (звіти, графіки розробки, повідомлення);

— контроль проведення формальних інспекцій та оглядів;

— аналіз і контроль проведення формального та приймального тестування (випробувань) програмного забезпечення.

З якістю асоціюються такі властивості програмного забезпечення, як захист, безпека, толерантність, зручність використання (зручний для читання, адаптації, прозорість виявлення помилок) та супроводження.

Залежно від специфіки програмних продуктів стандарти пропонують термінологію та склад атрибутів (показників) якості. Вони утворюють базові знання і визначають планування, проектування, аналіз, вимірювання та поліпшення якості.

Якість програмного забезпечення визначається набором загальних характеристик, що вказують на спроможність задовольнити вимоги замовника і всіх потенційних користувачів цього продукту включно із системними користувачами. На формулювання таких потреб спрямовані процеси інженерії вимог як складової частини програмної інженерії.

Характеристики якості визначають споживчі властивості і мають вартісний вираз, що включає оцінку витрат на процес розробки й експлуатації, оцінку економічних вигод застосування заданих програм порівняно з іншими засобами вирішення відповідного прикладного завдання, а також перспектив подальшого використання напрацьованого програмного забезпечення в умовах зміни середовища функціонування або оточення.

Вигода від експлуатації програми визначається комплексом оцінок, одержаних під час перевірки виконання визначених для неї функцій у реальних життєвих ситуаціях, зручностей використання порівняно з іншими засобами, а також часом обслуговування, відновлення після збоїв/відмов і підключення спеціального персоналу для проведення супроводження програмного забезпечення. Вартісна оцінка продукту включає й оплату персоналу.

Більшість характеристик, які визначають зручність, може бути оцінено в одиницях часу на освоєння персоналом використовуваної програми, на підготовку вхідних даних, реалізацію завдань на виконання, на аналіз вихідних даних і відновлення сил персоналу після роботи з програмним забезпеченням.

## 9.2. Організація робіт для досягнення якості програмного забезпечення

Досягнення необхідної якості програмного забезпечення здійснюється в рамках комплексної програми управління якістю [6], яка має включати організаційні й методологічні заходи з підготовки умов для аналізу якості програмного забезпечення, виконуваних спеціальною групою, а також шляхом проведення робіт з оцінки рівня якості програмного забезпечення при розробці, виготовленні, експлуатації і супроводженні. Аналіз та управління якістю включають ви-

рішення завдань планування, обліку, контролю і регулювання якості програмного забезпечення.

До організаційно-методичних заходів належать:

- визначення моделі якості, що відображає номенклатуру показників якості та їхніх значень для даного класу програмного забезпечення;
- вироблення основних етапів і положень, за якими буде проводитися збір даних для досягнення якості на етапах життєвого циклу;
- визначення методів оцінки якості продукту і документації на нього.

Ці заходи проводяться групою спеціалістів, що добре розуміються на питаннях забезпечення якості та його оцінки.

В основі управління якістю лежить модель якості, вибір номенклатури показників якої здійснюється за такими принципами:

- відкритість номенклатури;
- однозначність і незалежність атрибутів показників;
- узгодженість з базовими значеннями показників та з еталоном.

На початку розробки програмного забезпечення визначаються основні етапи життєвого циклу, на яких передбачається проведення робіт з аналізу стану якості й оцінки окремих атрибутів показників якості, що інтегруються в єдину оцінку по закінченню заключного етапу життєвого циклу програмного забезпечення.

Методи оцінки якості програмної продукції базуються на системі аналізу якості і на засобах перевірки супровідних документів проекту на дотримання стандартів та вимог замовника щодо досягнення визначеного рівня якості.

Для проведення оцінки якості програмного забезпечення попередньо визначаються базові значення показників для аналогу, прийнятого за еталон при розробці даного програмного забезпечення. Значення базових показників мають відбивати сучасний світовий рівень розробок.

На роль аналогів вибирається реальне програмне забезпечення того самого функціонального призначення, що й порівнюване, з такими самими основними параметрами, що й дана структура, і схожими умовами експлуатації.

## 9.3. Модель якості програмного забезпечення

Якість програмного забезпечення — це відносне поняття, що має сенс тільки при врахуванні реальних умов його застосування, тому вимоги, які пред'являються до якості програмного забезпечення, ставляться відповідно до умов і конкретної галузі їхнього застосування.

Якості програмного забезпечення відповідають три головних аспекти розробки: процеси життєвого циклу, програмний продукт, ефект від супроводження або впровадження (рис. 9.1).

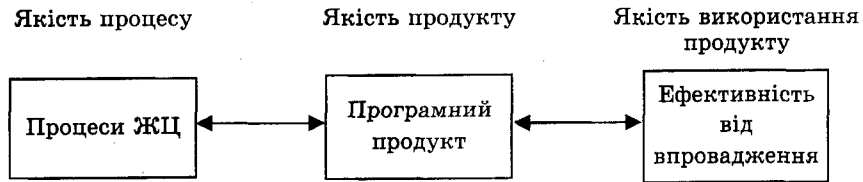


Рис. 9.1. Основні аспекти якості

Аспект, пов'язаний із процесами життєвого циклу, характеризується ступенем формалізації, достовірності і якості самих процесів ведення розробки програмного забезпечення. Він також характеризується ще й тим, як проводяться верифікація та валідація окремих отриманих вихідних результатів на проміжних процесах і на заключному.

Якість продукту цілком визначається процесами життєвого циклу. Ефект від впровадження отриманого програмного продукту значною мірою залежить від якості супроводження та знань особливостей його функціонування обслуговуючим персоналом.

Нині склалося певне уявлення про модель якості програмного забезпечення, що має чотири рівні деталізації [3—9].

**1-й рівень.** Визначення характеристик (показників) якості відображає погляд користувача на якість програмного забезпечення. Відповідно до стандарту ISO-9126 [3] визначено шість характеристик (або показників) та відповідно до гармонізуючого з ним вітчизняного стандарту ДСТУ 2850-94 [5] — моделі якості. Модель характеристик та атрибутів якості показано на рис. 9.2. Перелічимо визначені в джерелі [3] характеристики:

- 1) функціональність (functionality),
- 2) надійність (reliability),
- 3) зручність (usability),
- 4) ефективність (efficiency),
- 5) супроводжуваність (maintainability),
- 6) переносність (portability).

**2-й рівень.** Атрибути якості є визначенням складових кожної характеристики, що мають назву властивості, котрі деталізують аспекти певної характеристики. Кожній характеристиці якості відповідає набір атрибутів якості, за допомогою яких оцінюється досягнення цієї характеристики.

**3-й рівень.** Введення метрики як моделі вимірювання якості. Згідно з джерелом [7], метрика є комбінацією методу вимірювання атрибута та шкали вимірювання (тобто множини значень атрибута). Метрика — це система вимірів атрибутів, яку використовують під час оцінки продукту експертом та заповненні ним протоколу експертизи.

Для проведення оцінки атрибутів якості, досягнутих на етапах життєвого циклу (при перегляді документації, програм та резуль-

татів тестування програм) використовуються метрики з визначеною вагою оцінки для нівелювання результатів метричного аналізу сукупних атрибутів конкретного показника і якості в цілому. Атрибут якості обчислюється за допомогою однієї чи кількох методик оцінки на етапах життєвого циклу або комплексної оцінки готового продукту після завершення розробки.

**4-й рівень.** Оціночний елемент метрики задає спосіб оцінки значень окремих атрибутів програмного забезпечення і сприяє кількісній або якісній оцінці відповідного показника якості з урахуванням його ваги в системі оцінок якості.

Залежно від призначення, особливостей та умов супроводження програмного забезпечення вибираються найбільш важливі характеристики якості та їхні пріоритети.

Обрані для кожної характеристики атрибути та їхні пріоритети відображаються у вимогах на розробку.

Для програмних систем, при розробці яких у вимогах не зазначено пріоритетів характеристик якості, використовується пріоритет еталона — класу програмного забезпечення, до якого належить дана система.

Метрики показників якості являють собою сукупність оцінкових елементів і слугують для визначення ступеня відповідності кожної властивості програмного забезпечення заданому еталону. Кожна метрика може мати один або кілька елементів [3, 6, 7, 13, 14].

Оцінка якості програмного забезпечення за ієрархічною моделлю починається з нижнього рівня ієрархії, тобто із найелементарнішої властивості оцінюваного показника якості. На етапі проектування експерти встановлюють значення оцінкових елементів кожного атрибута аналізованого програмного забезпечення, включеного до вимог.

Якщо для даного типу програмного забезпечення у вимогах було визначено всі шість показників, то перелічені показники підсумовуються з урахуванням їхніх вагових коефіцієнтів. Унаслідок цього утворюється інтегральна оцінка рівня якості програмного забезпечення.

Нижче наводиться змістовний опис кожної характеристики з їхніми атрибутами (рис. 9.2).

### 9.3.1. Функціональність

Функціональність — це сукупність властивостей, які визначають спроможність програмного забезпечення виконувати в заданому середовищі перелік функцій відповідно вимогам до обробки і загальносистемним засобам.

Під функцією розуміють упорядковану послідовність дій для задоволення споживчих властивостей, замовлених користувачем. Функції бувають цільові (основні й допоміжні), які є основою, що забезпечує споживчі властивості у виконанні програмного забезпечення.

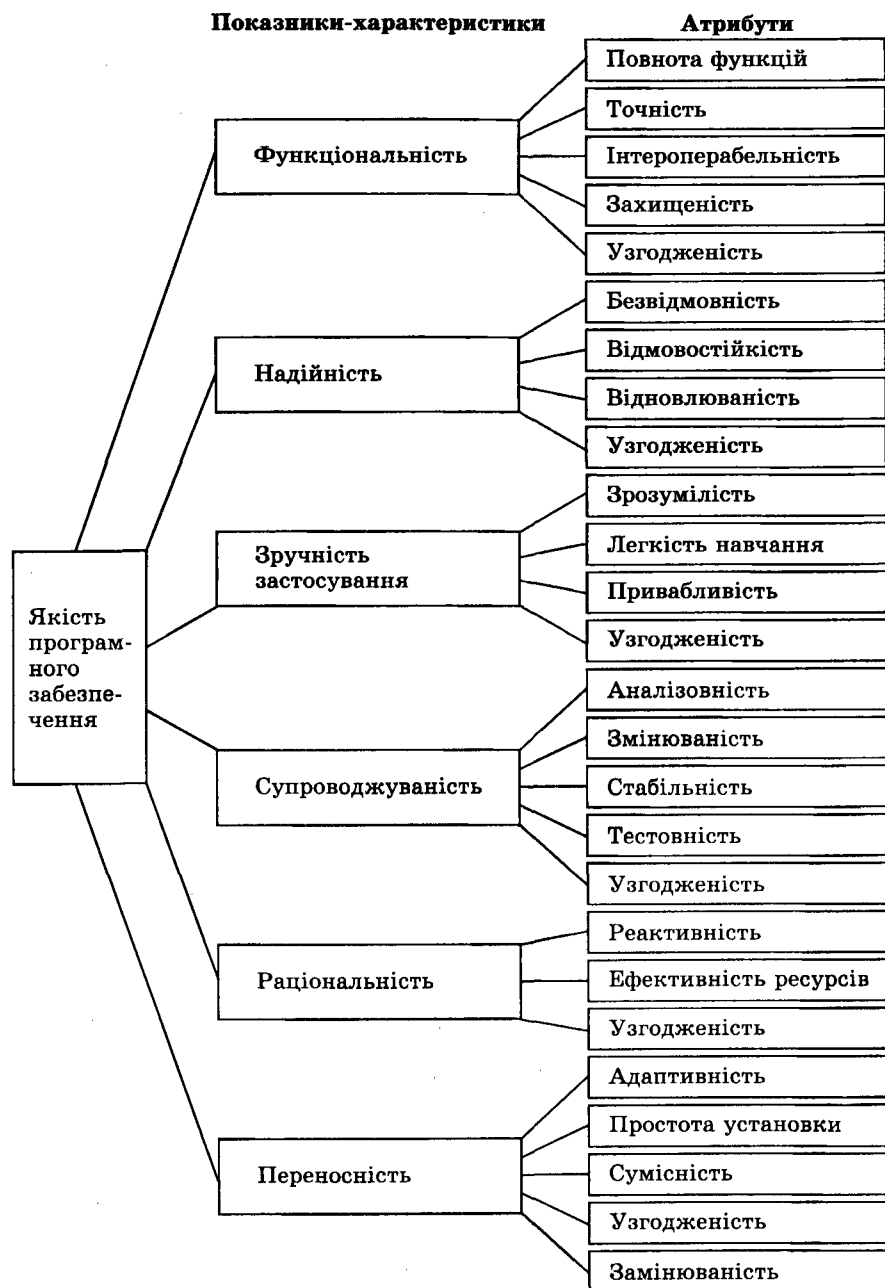


Рис. 9.2. Модель характеристик та атрибутів якості

Ця характеристика порівняно з іншими є, можна сказати, вирішальною. Відповідно до ISO/IEC 9126 функціональність відповідає на запитання: які споживчі властивості програмного забезпечення реалізовано без деталей того, як вони реалізуються. Інші атрибути якості слугують для того, щоб визначити ціну програмного забезпечення і можливості використання.

До атрибутів функціональності програмного забезпечення належать:

- функціональна повнота: атрибут, який показує ступінь достатності основних функцій для вирішення спеціальних завдань відповідно до призначення програмного забезпечення;

- правильність (точність): атрибут, який показує, як забезпечується досягнення правильних, теоретично правильних та погоджених результатів;

- інтероперабельність: атрибути, які вказують на спроможність програмного забезпечення взаємодіяти із спеціальними системами і середовищами, наприклад з мережевими;

- захищеність: атрибути, які вказують на можливість запобігати несанкціонованому доступу (випадковому або навмисному) до програм і даних;

- узгодженість: атрибут, який вказує на відповідність заданим стандартам, угодам, правилам, законам і розпорядженням.

### 9.3.2. Надійність

Надійність — це множина атрибутів, котрі вказують на спроможність програмного забезпечення перетворювати вхідні дані на результати за умов, що залежать від періоду часу (зношення й старіння програмного забезпечення не враховуються) [14—17]. Зниження надійності програмного забезпечення відбувається через помилки у вимогах, проектуванні і виконанні. Відмови та помилки залежать від засобу виробництва продукту і з'являються в програмах протягом заданого проміжку часу.

До підхарактеристик надійності програмного забезпечення належать:

- безвідмовність: атрибути, які визначають відсутність помилок або частоту відмов через наявність помилок у програмному забезпеченні;

- стійкість до помилок: атрибути, які вказують на забезпечення спроможності виконувати функції в аномальних умовах (збої апаратури, помилки в даних та інтерфейсах, порушення в діях оператора тощо);

- відновлюваність: атрибути, які вказують на спроможність програми до перезапуску для повторного виконання й відновлення даних після відмов;

- узгодженість: атрибут, який показує відповідність наявним стандартам, угодам, правилам, законам і розпорядженням.

Для багатьох систем (програм, діяльностей та апаратури) надійність є головною цільовою функцією реалізації важливих атрибутів,

що може визначатися по-різному. Іноді атрибути надійності виражаються через інші атрибути якості, якщо останні дозволяють досягнення високої якості програмного забезпечення.

Деякі типи систем (реального часу, радарні системи, системи безпеки, комунікація, медичне устаткування та інші) містять особливі вимоги до забезпечення високої надійності включно з такими атрибутами, як недопустимість помилок, безпека, захищеність і зручність застосування, а також достовірність як основний критерій надійності.

Надійність складних програмних систем істотно відрізняється від надійності апаратури. Носії даних (файли, сервер і под.) мають високу надійність. Записи на них можуть зберігатися тривалий час без руйнування. Чинник руйнування і старіння носіїв даних не відіграє істотної ролі в оцінці надійності програмного забезпечення.

Отже, надійність програмного забезпечення значною мірою залежить від числа залишених у процесі розробки на етапах життєвого циклу і не усунених помилок. У ході експлуатації помилки теж виявляються й усуваються. Якщо при виправленні помилок не вносяться нові, або принаймні, нових помилок вноситься менше, ніж усувається, то в ході експлуатації надійність програмного забезпечення безупинно зростає. Чим інтенсивніше проводиться експлуатація, тим інтенсивніше виявляються помилки і швидше зростає надійність програмного забезпечення.

Термін надійності програмного забезпечення використовується в ролі показника якості програмного забезпечення. Виникає необхідність уточнення ролі атрибутів надійності в системі показників (характеристик) якості.

Надійність є функцією від помилок, які залишилися в програмному забезпеченні після впровадження його в експлуатацію. Програмне забезпечення, яке не має помилок, є абсолютно надійним. Але для великих програм абсолютна надійність практично недосяжна. Невиявлені помилки, котрі залишилися, виявляють себе час від часу в певних умовах (на деякій сукупності вихідних даних) супроводження й експлуатації.

Надійність — єдиний кількісний показник якості, що характеризує властивість програмного забезпечення виявляти помилки, які залишилися в ньому, в процесі експлуатації при заданому наборі вхідних даних.

Для оцінки надійності програмного забезпечення [10—13] використовуються такі статистичні показники, як можливість та час безвідмовної роботи, можливість відмови і частота відмов. Оскільки тільки помилки в програмі розглядаються як причини відмов, що не можуть самоусунутися, то програмне забезпечення варто відносити до класу невідновлюваних систем. За основний критерій надійності в таких системах приймається тривалість роботи системи до першої відмови.

Для одержання цих характеристик потрібно провести статистичні випробування. Зауважимо, що при кожному прояві нової помилки, як правило, проводяться її локалізація і виправлення. Правду кажучи, набрана до цього статистика втрачає своє значення, тому що після внесення коректив налагоджена програма, по суті, є новою програмою з іншими характеристиками, відмінною від тієї, що до цього випробовувалася.

У процесі виправлення помилок показники надійності будуть весь час змінюватися, як правило, у бік поліпшення. Отже, їхня оцінка матиме тимчасовий і наближений характер. Тому виникає необхідність пошуку нових підхарактеристик, адекватних реальному процесу вимірювання надійності, таких, як залежність інтенсивності виявлених помилок від числа прогонів програми (тобто, від числа реалізацій) і залежність відмов від часу функціонування програмного забезпечення тощо.

Для систем з високим ступенем інтеграції, відмови в яких можуть знизити безпеку або захист, надійність — це спосіб визначення якості.

До чинників гарантії надійності належать:

- загроза — прояв нестійкості, що порушує безпеку системи;
  - ризик як сукупність загроз, що можуть призвести до несприятливих наслідків і збитку, а також до змін стану системи або середовища;
  - аналіз ризику як вивчення загрози або ризику, їхня частота і наслідки;
  - цілісність — спроможність системи зберігати стабільність роботи і не мати ризику;
  - рівні цілісності програмного забезпечення як необхідні його властивості, що утримують ризик системи в рамках прийнятих обмежень.
- Ризик перетворює і зменшує надійність, тому що виявлені помилки можуть призвести до загрози, якщо відмови носять частотний характер.

### 9.3.3. Зручність застосування

Зручність — це множина атрибутів, котрі вказують на необхідні або сприятливі умови для використання програмного забезпечення визначеним колом користувачів. Під використанням розуміють як діалогове спілкування з користувачем, так і недіалогове функціонування й одержання результатів виконання в різноманітних середовищах, в яких можуть проводитися підготовка й опрацювання результатів. Це одна з характеристик зручності застосування.

У міжнародному стандарті [3] зручність застосування визначено як специфічну множину атрибутів програмного продукту з властивостями ергономічності.

До підхарактеристик зручності застосування програмного забезпечення належать:

— зрозумілість: атрибути, які визначають зусилля, необхідні для розпізнавання логічних концепцій та умов застосування їх;

— легкість навчання: атрибути, які визначають зусилля, затрачені користувачами, щоб вивчити умови використання, наприклад, операційний контроль, впровадження, висновки, діагностику, а також процедури, правила і документацію для використання програмного забезпечення;

— оперативність: атрибути, які характеризують якість реакції системи на зусилля користувача при виконанні операцій та операційного контролю;

— узгодженість: атрибут, який показує відповідність розробки до вимог діючих стандартів, угод, правил, законів і розпоряджень.

#### 9.3.4. Ефективність

Ефективність забезпечується множиною атрибутів, які показують взаємозв'язок між рівнем виконання програмного забезпечення і кількістю використовуваних ресурсів у початкових умовах. До ресурсів належать засоби, апаратура, інші використовувані продукти, матеріали (наприклад, папір для друкувального пристрою), послуги штатного обслуговуючого персоналу тощо.

До підхарактеристик ефективності програмного забезпечення належать:

— час реакції (реактивність) — атрибути, які показують час відгуку, опрацювання і виконання функцій програмного забезпечення;

— ефективність ресурсів — атрибути, які показують кількість і тривалість використовуваних ресурсів при виконанні функцій програми в програмному забезпеченні;

— узгодженість: атрибут, який показує відповідність даного атрибута заданим стандартам, правилам і розпорядженням.

#### 9.3.5. Супроводжуваність

Супроводжуваність — це множина властивостей, які вказують на зусилля, котрі необхідно витратити на проведення модифікацій, таких як коригування, удосконалення й адаптацію програмного забезпечення зі зміною середовища, вимог або функціональних специфікацій. Тобто, модифікацією вважаються будь-які зміни, які проводяться з метою поліпшення або додавання деяких особливостей функціонування програмного забезпечення (див. главу 8).

До підхарактеристик супроводжуваності програмного забезпечення належать:

— аналізованість: показник, який визначає необхідні зусилля для діагностики випадків відмов або ідентифікації частин, що будуть модифікуватися;

— змінюваність — показник, який визначає зусилля, що витрачаються на модифікацію, видалення помилок або внесення змін у зв'язку з помилками або новими можливостями середовища функціонування;

— стабільність: атрибути, які вказують на ризик модифікації;

— тестованість: атрибути, які вказують на зусилля для проведення валідації та верифікації з метою виявлення помилок у програмах і невідповідностей програм вимогам, а також наступної модифікації програмного забезпечення та його атестації;

— узгодженість: атрибут, який вказує на відповідність даного атрибута атрибутам у стандартах та угодах, правилам і розпорядженням.

#### 9.3.6. Переносність

Переносність — це множина показників, котрі вказують на здатність програмного забезпечення пристосовуватися до роботи при зміні середовища виконання. Середовище може бути організаційним, апаратним і програмним. Тому перенесення програмного забезпечення може бути пов'язаним із сукупністю дій, спрямованих на забезпечення функціонування програмного забезпечення в середовищі, відмінному від того середовища, в якому воно створювалося, з урахуванням нових програмних, організаційних і технічних можливостей.

До підхарактеристик переносності програмного забезпечення належать:

— адаптивність: атрибут, який характеризує зусилля, необхідні для адаптації до спеціальних середовищ без виконання інших дій або засобів підтримки програмного забезпечення;

— настроюваність (простота інсталяції): показники необхідних зусиль для запуску або інсталяції даного програмного забезпечення в певному середовищі;

— сумісність: показники, які визначають можливість використання спеціального програмного забезпечення в середовищі чинного програмного забезпечення;

— узгодженість: атрибути, які вказують на відповідність стандартам або домовленостям із забезпечення переносності;

— інтегровуваність: атрибути, які забезпечують можливість взаємозамінювання при спільній роботі з іншими програмами з необхідною інсталяцією або адаптацією програмного забезпечення.

Таблиця 9.1 містить стислий опис наведених вище атрибутів.

### 9.4. Методи визначення значень показників якості

Для визначення значень показників якості відповідно до ДСТУ 2844-94 [4] застосовуються такі методи: вимірювальний, реєстраційний, розрахунковий та експертний (а також комбінації цих методів).

*Вимірювальний метод* базується на використанні вимірювальних і програмних засобів для одержання інформації про кількісні характеристики програмного забезпечення, наприклад, про визначення обсягу, числа рядків коду або операторів, кількості гілок у програмі, кількості точок входу (виходу), реактивності тощо.

**Таблиця 9.1. Стисла характеристика показників якості програмного забезпечення**

Показник	Опис властивостей показника
Функціональність	Група властивостей, що зумовлює його спроможність виконувати визначений перелік функцій, які задовольняють потреби відповідно до призначення.
Надійність	Група властивостей, що зумовлює спроможність програмного забезпечення зберігати працездатність і перетворювати вхідні дані в результат за встановлений період часу, навіть за відмов внаслідок внутрішніх дефектів і порушення умов його застосування.
Зручність застосування	Сукупність властивостей програмного забезпечення для передбачуваного кола користувачів, що відображають легкість його освоєння та адаптації до умов експлуатації, які змінюються, легкість підготовки даних, зрозумілість результатів, зручність внесення змін у програмну документацію та в програми.
Супроводжуваність	Група властивостей, які визначають зусилля, необхідні для виконання та внесення змін, модифікації, а також атестації програмного забезпечення, що модифікується.
Раціональність	Група властивостей, що характеризує ступінь відповідності використовуваних ресурсів середовища функціонування рівню якості (надійності) функціонування при заданих умовах застосування програмного забезпечення.
Переносність	Група властивостей, що забезпечує пристосованість до переносу з одного середовища функціонування в інші, зусилля для перенесення й адаптації програмного забезпечення до нового середовища.

*Реєстраційний метод* базується на підрахунку деяких подій, моментів часу, числа збоїв або відмов, початку і кінця роботи програмного забезпечення в процесі його виконання.

*Розрахунковий метод* базується на статистичних даних, зібраних під час проведення випробувань, експлуатації і супроводження програмного забезпечення. Розрахунковими методами оцінюються показники надійності, точності, стійкості, реактивності тощо.

*Експертний метод* здійснюється групою експертів-спеціалістів, компетентних у вирішенні даного завдання або типу програмного

забезпечення. Їхні рішення базуються на досвіді та інтуїції, а не на безпосередніх результатах розрахунків чи експериментів. Цей метод реалізується шляхом перегляду програм, кодів, супровідних документів, унаслідок чого визначається якісна оцінка створеного продукту. Для цього встановлюються контрольовані ознаки, що корелюють з одним або кількома показниками якості, які включаються до опитувальних карт експертів. Метод застосовується для оцінки таких показників, які характеризують здатність до аналізування, документування, структурування, узгодження.

Для оцінювання значень показників якості залежно від особливостей використовуваних ними властивостей, призначення, засобів їхнього визначення використовуються шкали:

- метрична (1.1 — абсолютна, 1.2 — відносна, 1.3 — інтегральна);
- порядкова (рангова), що дозволяє ранжувати характеристики шляхом порівняння з опорними;
- класифікаційна, що характеризує тільки наявність або наявності здатності до аналізованості в оцінюваному програмному забезпеченні.

Показники, які мають метричні шкали, називаються кількісними, а порядкові і класифікаційні — якісними.

## 9.5. Метрики якості програмного забезпечення

Поки що в програмній інженерії ще не склалась остаточно система метрик. Однак діють різноманітні підходи й методи щодо складу метрик та вимірювання їх. Про важливість цієї проблематики писав лорд Кельвін: “Коли ми не можемо виміряти і висловити все в числах — наші знання обмежені і недостатні”.

У галузі розробки програмного забезпечення система вимірювання є необхідною і включає метрики та моделі вимірів [8,19]. Ця система вимірів використовується для кількісних оцінок програмного забезпечення і досягнення розуміння, контролю й поліпшення якості продукту.

Для визначення вимог до програмного забезпечення задаються відповідні їм зовнішні характеристики та їхні підхарактеристики (показники), що стосуються різних аспектів функціонування й керування продуктом у заданому середовищі. Для заданого набору характеристик якості програмного забезпечення у вимогах визначаються відповідні метрики, моделі їхньої оцінки та діапазон значень мір для виміру підхарактеристик або атрибутів якості.

Метрика є атрибутом продукту, що вимірюється за однією із шкал, зазначених вище, і включає множину допустимих значень цього атрибута для кожної характеристики якості.

Відповідно до стандарту [5] метрики визначаються за моделлю виміру атрибутів програмного забезпечення на всіх етапах життєвого

циклу (проміжна, внутрішня метрика) і особливо на етапі тестування або функціонування (зовнішні метрики) продукту.

Зупинимось на класифікації метрик програмного забезпечення, на правилах для проведення метричного аналізу прикладних застосувань та на процесі вимірювання їхніх атрибутів.

**Типи метрик.** Є два типи метрик:

1) метрики програмного продукту, що використовуються для виміру його характеристик-властивостей;

2) метрики процесу, що використовуються для виміру властивостей процесу створення продукту.

### 9.5.1. Метрики програмного продукту

Метрики програмного продукту включають:

- зовнішні метрики властивостей продукту, видимих користувачеві;
- внутрішні метрики властивостей, видимих тільки команді розробників.

**Зовнішні метрики** продукту включають:

— метрики надійності продукту, які слугують для визначення числа дефектів;

— функціональні метрики, за допомогою яких встановлюється наявність і правильність реалізації функцій у продукті;

— метрики супроводження, за допомогою яких вимірюються ресурси продукту (швидкість, пам'ять, середовище);

— метрики придатності продукту, які сприяють визначенню ступеня доступності для вивчення і використання;

— метрики вартості, які визначають вартість створеного продукту.

**Внутрішні метрики** продукту включають:

— метрики розміру, які забезпечують вимірювання продукту за допомогою його внутрішніх характеристик;

— метрики складності, за допомогою яких визначається складність продукту;

— метрики стилю, які слугують для визначення підходів і настанов у створенні програм та окремих компонентів продукту і його документів.

Внутрішні метрики проектуються для підвищення контролю якості виробництва продукту і є релевантними щодо зовнішніх метрик.

Зовнішні і внутрішні метрики задаються на етапі формування вимог до програмного забезпечення і є предметом планування шляхів досягнення якості кінцевого програмного продукту.

Метрики продукту часто описуються комплексом моделей для надання різноманітних властивостей та значень моделі або для прогнозування їх. Вимірювання проводяться, як правило, після калібрування метрик у ранніх проектах.

Метрики продукту може бути запропоновано для кожної стадії життєвого циклу. Загальною мірою є ступінь трасування, що визначається як число вимог, які простежуються за допомогою моделі сценаріїв. Прикладом метрик можуть бути:

- число вимог;
- число сценаріїв і діючих осіб (акторів);
- число об'єктів, включених до сценарію, і локалізації вимог у кожному сценарії;
- число параметрів і операцій об'єкта та ін.

Спеціальною мірою може виступати рівень використання повторних компонентів, яка визначається як відношення розміру продукту, виготовленого з готових компонентів, до розміру системи в цілому. Ця міра використовується для визначення вартості і якості програмного забезпечення. Прикладами метрик повторних компонентів є:

- загальне число об'єктів і число повторно використовуваних;
- загальне число операцій, повторно використовуваних і нових;
- число класів, що успадковують специфічні операції;
- число класів, від яких залежить даний клас;
- число користувачів класу або операцій та ін.

Для оцінки загальної кількості деяких вимірів часто використовуються середньостатистичні метрики, наприклад такі, як середнє число операцій у класі, середнє число спадкоємців класу або операцій класу та ін.

Як правило, міри оцінки є достатньо суб'єктивними і залежать від знань експертів, котрі виконують кількісне оцінювання елементів програмного продукту.

### 9.5.2. Метрики процесів

Метрики процесів включають:

- метрики вартості, які вимірюють затрати праці на створення продукту або архітектури проекту з урахуванням таких властивостей, як оригінальність розробки, підтримка, документація;
- метрики досягнень, як частина метрик вартості і тих, що вимірюють вартість праці розробника в людино-днях чи місяцях;
- метрики ненадійності процесу, які встановлюють шляхом прогнозування експертів числа невиявлених дефектів під час проектування;
- метрики повторюваності, які встановлюють, як у даному процесі використовуються раніше виготовлені елементи.

Метрики проекту включають властивості заданих основних понять предметної галузі, які підлягають вимірюванню. Вимірюються переважно внутрішні й зовнішні атрибути програм.

Як метрики процесу можуть використовуватися: час розробки, число помилок, знайдених під час тестування. Зокрема стосовно сценарного підходу, викладеному в главі 3, використовуються такі метрики процесу:

- загальний час розробки й окремо час для кожної стадії;
- час модифікації моделей;
- час виконання робіт у процесі;
- число знайдених помилок під час ревізії;
- вартість забезпечення якості;
- вартість проведення процесу розробки.

Проте іноді здійснюються вимірювання зайвих величин у проєкті, наприклад число знаків ASCII у програмі, хоча це число нікого не цікавить, у тому числі й менеджера проєкту, якого більше цікавить число підпрограм, окремих компонент проєкту та їхніх атрибутів. Саме це лежало в основі вимірювання бібліотеки мови програмування [8,13], для якої було розглянуто 150 класів об'єктів і 1850 властивостей. Для них знайдено середній коефіцієнт використання класів відповідно 0.4 і 3.

Прикладом широко використовуваних зовнішніх метрик програм є метрики Холстеда — метричні характеристики програм, що виявляються на основі статичного представлення програми конкретною мовою програмування. Як вимірні атрибути алгоритму розглядаються: число входжень операндів, які трапляються найчастіше, та операторів; довжина опису програми як сума числа входжень всіх операндів і операторів.

Ці атрибути є джерелом для одержання таких метричних співвідношень, як час роботи і час програмування (для обчислення показника трудомісткості розробки), обчислення рівня програми (показник її структури та якості), рівня мови програмування (показник рівня абстракції засобів мови й орієнтації його на дану проблему) тощо.

### 9.5.3. Вимірювання метрик якості

Вимірювання властивостей та характеристик якості на етапах життєвого циклу сприяє одержанню якісного продукту і відповідає меті та завданням стандарту [7]. Це є процес застосування метрик у числовій і нечисловій формах на всіх етапах життєвого циклу. Метрики включають атрибути (зовнішні і внутрішні), які слугують для виміру як окремих показників якості, так і якості в цілому. Крім того, вони сприяють поліпшенню результатів, отриманих у процесі розробки програмного забезпечення.

Основним кількісно вимірюваним атрибутом якості програмного продукту є надійність, що характеризує цей продукт щодо його функціонування без збоїв та помилок.

Нині спостерігається тенденція до вимірювання атрибутів проєкту (наприклад, ступінь розподілу зусиль за фазами проєкту), які визначають структуру створюваного програмного забезпечення та його елементів (наприклад, ступінь повторного використання компонент й оцінка вартості робіт з їхнім застосуванням).

Метрики якості використовуються також для оцінки ступеня тестовності після проведення випробувань програмного продукту на множині тестів з метою перевірки безвідмовної роботи, виконання функцій, зручності застосування інтерфейсів користувача, баз даних тощо. З метою вимірювання вибираються сукупності мір для оцінки атрибутів програмного забезпечення та відповідні їм метрики.

Використання метрик має проблеми, пов'язані з великою різноманітністю їх, і, крім того, вони найчастіше не корелюють із стандартами, властивостями та характеристиками використовуваної для даного типу програмного забезпечення моделі якості.

У цілому жодний з показників не може слугувати надійною базою для оцінювання програми. Розглянемо два характерних приклади: показник надійності і показник “складності” як довжина програми. Перший належить до характеристик користувача і його може бути виміряно при відносній готовності програми в процесі її функціонування. Другий показник є технічним і характеризує, до певної міри, вміння відображати позицію розробника програми.

Наробіток на відмову, як атрибут надійності, визначає середній час між виникненнями загроз, що порушують безпеку, і замінює важко вимірювану оцінку збитку, що завдається відповідними загрозами. Підхід до виміру цієї характеристики запозичено з оцінки якості технічних засобів. Наприклад, один користувач діалогової програми звик використовувати меню, вказуючи варіанти за допомогою клавіатури, а іншому зручніше працювати правою кнопкою “миші”. У першого користувача програма працює стабільно, у другого виникають аварійні ситуації, пов'язані з тим, що під час натискання кнопки “миші” він активізує випадковий рядок меню, чим викликає відмову.

Дуже часто оцінка програми проводиться за числом рядків. При зіставленні двох програм, котрі реалізують одне прикладне завдання, перевага, як правило, віддається короткій програмі, тому що її, напевно, створює більш кваліфікований персонал, у ній менше прихованих помилок і її легше модифікувати. За вартістю вона дорожча, хоча на її налагодження та модифікацію витрачається більше часу.

Довжину програми можна використовувати як допоміжну метрику для порівняння програм при умові рівної кваліфікації розробників, єдиного стилю розробки і загального оточення.

Мета оцінювання якості на основі моделі якості полягає в тому, щоб порівняти отримані результати із встановленими вимогами до програмного забезпечення і визначити відповідність за кожною реалізованою характеристикою.

На основі виміру кількісних характеристик і проведення експертизи якісних показників із застосуванням вагових коефіцієнтів, які нівелюють показники, обчислюється підсумкова оцінка якості продукту шляхом підсумовування результатів за окремими показниками.

ми і зіставлення їх з іншими показниками програмного забезпечення (вартість, час, ресурси і под.).

Тобто, під час оцінювання окремого показника за допомогою оцінних елементів враховуються  $k$  — ваговий коефіцієнт,  $j$  — показник,  $i$  — атрибут. Наприклад, у ролі  $j$ -показника візьмо переносність. Тоді цей показник буде розраховуватися за п'ятьма атрибутами ( $i = 1, \dots, 5$ ), наведеними на рис. 9.2, причому кожний з них буде множитися на відповідний коефіцієнт  $k$ .

Унаслідок цього всі метрики  $j$ -атрибути підсумовуються й утворюють  $j$ -показник якості. Коли всі атрибути перелічено за кожним із шести показників якості (див. табл. 9.1), проводиться сумарна оцінка окремого показника, а потім й інтегральна оцінка якості з урахуванням вагових коефіцієнтів усіх показників програмного забезпечення.

В остаточному підсумку результат оцінки якості є критерієм ефективності та доцільності використаних методів проектування, інструментальних засобів і методик оцінювання результатів створення програмного продукту на стадіях життєвого циклу.

## 9.6. Аналіз якості програмного забезпечення

Під час аналізу якості розглядаються характерні риси програмного забезпечення, які корисні для поточного проекту та проектів, котрі будуть виконуватися. Наприклад, один замовник хоче, щоб модулі або об'єкти були надійними, а інші хочуть, щоб, по можливості, уважніше розглядалися всі запропоновані основні вимоги до якості програмного забезпечення.

Основними напрямками забезпечення якості є:

- надійність об'єкта,
- трасування вимог,
- зв'язок-об'єднання.

У розробці якісного програмного забезпечення є спеціальні ситуації, що потребують визначення додаткових критеріїв, які не відіграють ролі для інших видів програмного забезпечення.

Для деяких типів систем, наприклад систем реального часу, потрібно забезпечити реактивність системи в заданий проміжок часу, а програмне забезпечення має цю вимогу гарантувати. Програмне забезпечення є багатогранним, вимоги до нього відображаються в його можливостях, тому дуже важко визначити загальні еталони для таких типів систем, як системи, ґрунтовані на знаннях і штучному інтелекті, системи взаємодії і користувацького інтерфейсу, інформаційні системи, системи клієнт-сервер.

При плануванні аналізу якості для спеціальних продуктів розглядаються такі чинники:

1) середовище системи, в якому програмне забезпечення буде працювати;

2) вимоги до функцій системи;

3) стандарт, який використовується під час розробки компонентів системи;

4) стандарти розробки програмного забезпечення в цілому;

5) стандарти, використовувані для забезпечення якості програмного забезпечення;

6) методи і засоби, застосовувані для розробки програмного забезпечення;

7) методи і засоби оцінювання якості;

8) бюджет, штати, організація робіт, плани і графіки;

9) користувачі системи.

Деякі атрибути якості (безпека, захист) можуть конфліктувати один з одним або з іншими функціональними вимогами, що впливають на вартість продукту. Для ідентифікації потенційно важких місць у розробці програмного забезпечення використовуються різноманітні техніки планування. Аналіз загрози і ризику в проблемі ідентифікації служить для організації безпеки та захисту програмного забезпечення і даних. На цей час визначилися технології контролю та змінювання продукту.

Наслідком можливих відмов і помилок у системі є низький ступінь якості вимог.

Планування аналізу і перевірка якості вимог на початкових етапах дають упевненість у тому, що вони не суперечать вимогам системи та стандартам, а також ідентифікації додаткових ресурсів і технологій, необхідних для створення продукту.

Оскільки якість є споживчою властивістю програмного продукту, то до аналізу якості входять процеси визначення ризику, встановлення його властивостей та аналіз роботи продукту для пошуку дефектів. Аналіз якості проводиться за каркасом системи, що сприяє попередженню появи вихідних дефектів у роботі продукту. Проект має ризик, якщо в ньому є помилка.

Проблема керування ризиком включає ідентифікацію поняття ризик для аналізу програм. Аналіз якості програм виконується за допомогою тестів для виявлення дефектів перед тим, як продукт почне працювати.

Термін *продукт* означає будь-який результат процесу (так званий артефакт) проектування програмного забезпечення, а також будь-якого процесу, що використовується для побудови остаточного продукту. Згідно з цим, продуктом можуть бути:

- специфікації вимог для однієї програми як компоненти системи;
- спроектований модуль, коди, тестові документи;
- звіти аналізу якості.

Роль аналізу якості для побудови якісного продукту полягає в тому, щоб забезпечити керування вимірюванням якості вихідного результату

процесів розробки програмного забезпечення. Він включає оцінку діяльності із забезпечення якості шляхом верифікації та валідації [21], що може виконуватися або в статичній, або в динамічній формі виконання програми. Ці дії можуть виконати не лише розробники, а й інші люди, наприклад, індивідуальні проектувальники, або групи проектувальників.

Процеси верифікації і валідації визначають розробку продуктів при послідовному конформуванні вимог замовників із системами специфікацій та моделей, утворюваних впродовж життєвого циклу, і забезпечують одночасне дослідження одержуваних вихідних результатів на всіх проміжних підпроцесах перевірки задоволення системних вимог.

Для цих процесів є багато класифікацій. Для одних із них техніку верифікації та валідації використовують інженери програмного забезпечення, котрі створюють продукти, для інших — процеси верифікації і валідації мають бути з певним рівнем незалежності від розробки і, нарешті, для третіх — рівень цілісності продукту визначає ступінь незалежності продукту. Для аналізу якості й проведення верифікації та валідації створюються плани діяльності, техніки і вимірювання.

## 9.7. Стандарти підтримки якості програмного забезпечення

Для забезпечення якості програмного забезпечення є стандарти [1—6]. Стандарт [1] орієнтований на виконання двох основних завдань під час розробки програмного забезпечення:

- 1) гарантія (підтвердження) якості програмного забезпечення;
- 2) інженерія якості програмного забезпечення (Software Quality Engineering — SQE).

Процес гарантії якості призначений для забезпечення оцінювання якості продуктів і процесів проекту на етапах життєвого циклу, для втілення специфікованих вимог й усунення ризику під час виконання планів проектування. Цей процес планується як сукупність систематичних діяльностей для побудови якісного програмного забезпечення за допомогою розгляду чинників, адекватних вимогам і визначеним у вимогах властивостям. Крім того, на нього покладається керування якістю продуктів та методами ризику [20].

**Гарантія якості** — це підтвердження відповідності програмного забезпечення стандартам та процедурам досягнення якості (забезпечення якості) шляхом порівняльної оцінки показників, визначених у стандарті якості й отриманих у процесі розробки. Підтвердження проводиться на етапах життєвого циклу і являє собою такі дії:

- керування розробкою з гарантією якості;
- плани керування ризиком, конфігурацією та проектом;
- створення документації та звітних документів підтвердження якості програмного забезпечення.

- Ці дії виконує група забезпечення якості. Вони ґрунтуються:
  - на аналізі розроблюваної проектною документації;
  - на перегляді процесів розробки;
  - на керуванні та метричному аналізі властивостей і показників (характеристик) якості;
  - на приймальних випробуваннях шляхом тестування й оцінки ступеня тестовності та надійності роботи програмного продукту.
- Структуру й зміст планування аналізу якості визначає стандарт [2], а зазначена група провадить оцінку дотримання стандартів на етапах життєвого циклу.

**Інженерія якості** програмного забезпечення полягає в оцінці показників якості (надійності й супроводжуваності) на основі вимог до якості, спроможності протестувати програмне забезпечення, кількісно його виміряти і зіставити із зазначеними вимогами. Мета інженерії якості полягає в гарантованому створенні надійного і якісного програмного забезпечення із застосуванням таких дій:

- аналіз та ідентифікація обумовлених вимог до якості програмного забезпечення;
- збір метричних даних, аналіз їх і зіставлення із заданими вимогами до якості.

Ці дії не становлять самостійного процесу, а є невід’ємною складовою частиною процесів розробки програмного забезпечення на етапах ЖЦ.

Інженерія якості включає такі основні завдання:

- визначення методів та інструментів для проведення процесів розробки й оцінювання програмного забезпечення та показників якості;
- збір під час інженерної розробки програмного забезпечення статистичних даних, визначених стандартами, методиками і метриками для проміжної й остаточної кількісної оцінки окремих показників якості та їхньої відповідності заданим вимогам;
- проведення іспитів програмного забезпечення шляхом тестування, оцінки ступеня тестованості та нагляду за поведінкою і безпомилковим виконанням програмного забезпечення.

### 9.7.1. Конкретизація моделі якості

Модель якості, розглянута в п. 9.3, визначає загальні принципи атестації якості. Для конкретизації розробки необхідно визначити конкретний склад актуальних для неї характеристик, їхніх атрибутів, надати їм відповідно вимогам рейтингів тощо.

На вибір моделі якості і відповідних метрик впливають такі чинники:

- орієнтація на прикладну галузь, для якої необхідно визначити відповідні характеристики, встановити їхню вагу і відобразити думку користувачів на створюваний програмний продукт для цієї галузі;

- методологія проектування й інструментальні засоби (структурні або об'єктно-орієнтовані методи), які породжують множину метрик, що можуть впливати на характеристики якості;

- досвід і знання розробників програмної продукції, які будуть вести збір та реєстрацію помилок, а також вимірювання як окремих характеристик, так і якості програмного забезпечення в цілому;

- складність проєктованої системи (інформаційна, обчислювальна, логічна тощо), її розмір, який може вимагати декомпозиції системи на окремі компоненти, для яких має бути визначено вимоги до їхньої якості з єдиним механізмом оцінки;

- обрана модель якості, яка має орієнтуватися на сумісність інтересів замовника і розробника, а також на стратегію вимірювання окремих властивостей та перевірку відповідності отриманих характеристик якості заданим у вимогах, а також на класифікацію ризиків та їхні оцінки;

- тип моделі життєвого циклу (спіральна, водоспадна тощо) для керування якістю на всіх етапах життєвого циклу з орієнтацією на виміри окремих підхарактеристик якості; співвідношення між стандартними характеристиками якості і вибір головного базового атрибута (який може придушувати інші). Наприклад, для систем реального часу головною характеристикою є надійність, а для інформаційних систем — функціональність, повнота даних тощо.

### 9.7.2. Керування якістю

Керування якістю програмного забезпечення (Software Quality Management — SQM) здійснюється за допомогою комплексу засобів для керування політикою, метою і відповідальністю за досягнення якості програмного забезпечення. Найчастіше воно виконується як системна діяльність спеціальної групи якості з планування, оперативного керування, забезпечення й оцінки якості програмного забезпечення.

Планування якості — це діяльність, спрямована на визначення мети та вимог до якості і застосування елементів системи якості. Воно охоплює ідентифікацію, класифікацію й оцінку якості, а також визначення завдань, вимог до якості і штрафних санкцій за недотримання вимог. Планування включає моніторинг (відслідкування та контроль), складання календарного графіка для проведення аналізу стану розробки і послідовного вимірювання спланованих показників та критеріїв на стадіях життєвого циклу.

Оперативне керування включає методи і види діяльності оперативного характеру для поточного керування процесом, усунення короткострокових причин незадовільного функціонування на етапах життєвого циклу.

Забезпечення якості полягає у здійсненні необхідної діяльності для створення впевненості в тому, що об'єкт розробки задовольняє

вимоги до якості. Мета забезпечення якості може бути внутрішньою й зовнішньою. Внутрішня мета — це створення впевненості в керівника проєкту, що якість забезпечується. Зовнішня мета — це створення впевненості в користувача, що необхідної якості досягнуто і результатом є якісне програмне забезпечення.

**Системи якості програмного забезпечення.** Система якості (Quality systems — QS) [3—5] — це набір організаційних структур, методик, заходів, процесів і ресурсів для здійснення керування якістю. Система якості в організації розробників програмних систем призначена для задоволення потреб керування процесами життєвого циклу. Ці потреби ширші, ніж конкретні вимоги до програмного продукту певного споживача. Мета цієї системи — одержання продукту високої якості. Для досягнення необхідного рівня якості програмного забезпечення в міжнародній практиці програмування застосовуються два підходи. Один з них орієнтований на кінцевий програмний продукт, а другий — на процес створення продукту.

У підході, орієнтованому на продукт, акцент робиться на оцінюванні якості шляхом його випробування. Цей підхід базується на припущенні, що чим більше виявлено й усунуто помилок у продукті під час випробувань, тим менше їх залишилося і тим вище якість продукту. При цьому є такі недоліки:

- 1) усунення помилок у готовому продукті на етапі випробувань обходиться в десятки разів дорожче, ніж вчасне запобігання або усунення помилок на ранніх етапах життєвого циклу програмного забезпечення;

- 2) коли немає методів і засобів випробування програмного забезпечення, то це заважає гарантувати виявлення у випробовуваних програмах всіх помилок і всіляких відмов.

У другому підході акцент робиться на вживанні заходів для запобігання, оперативного виявлення й усунення помилок у програмному забезпеченні, починаючи з початкових етапів життєвого циклу. Для нього визначаються відповідальність виконавців, погоджуються плани і процедури забезпечення якості програмного забезпечення. Цей підхід тепер є загальноновизнаним, його покладено в основу концепції якості програмного забезпечення міжнародних організацій ISO/IEC і реалізовано в серії міжнародних стандартів та проєктів.

Основними стандартами в галузі якості продукції є стандарти ISO/IEC серії 9000. На їхній основі розробляються і вводяться в дію стандарти, орієнтовані на різноманітні типи програмних систем. Такі стандарти подано в джерелах [1—3, 7]. Суть їх полягає у наданні рекомендацій організаціям-розробникам програмного забезпечення створити систему якості. Загальну схему стандарту організації системи керування якістю згідно із джерелом [2] наведено на рис. 9.3.

Наявність в організації, що розробляє і поставляє замовникам програмне забезпечення, системи якості відповідно до вимог зазначе-

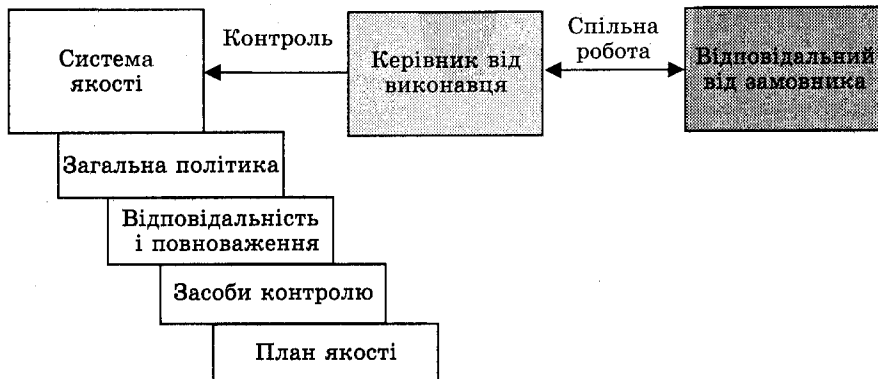


Рис. 9.3. Вимоги стандарту до організації системи якості

них стандартів найчастіше є умовою для розміщення солідних замовлень і, відповідно, довіри до розробника.

За рубежом багато фірм, котрі випускають готову продукцію, в обов'язковому порядку створюють системи якості, що забезпечує їм конкурентоспроможність продукції. Наприклад, японські фірми "Toyota", "Sony" та інші мають спеціальні відділи, які відповідають за погодження розробок з державними стандартами системи якості та її функціонування для виробництва кожного нового виду продукції.

Система якості має орієнтуватися на попит на продукцію, що випускається, на новітні технологічні досягнення для розробки нового виду продукції і суворий контроль усіх ланок виробництва включно з постачанням готових деталей та вузлів.

Природно, що без відповідних служб не буде конкуренції організацій — виробників вітчизняного програмного забезпечення на світовому ринку. Тож, щоб зарадити цьому, потрібно сконцентрувати зусилля на освоєнні передового міжнародного досвіду в організації контролю якості на основі міжнародних і національних стандартів, а також нормативних і методичних документів, які регламентують процес керування якістю програмного забезпечення для всіх категорій розробників і користувачів програмної продукції.

Такі заходи дадуть позитивний результат, якщо керівники всіх рівнів у галузі програмної індустрії почнуть здійснювати визначені кроки для створення системи забезпечення якості програмного забезпечення і вкладуть значні трудові й матеріальні (в тому числі й фінансові) ресурси, що повернуться досягненням якості.

**Сертифікація програмного продукту.** Під сертифікацією програмного продукту розуміють процес (здійснюваний як дія третьої сторони), який засвідчує відповідність ідентифікованої програмної продукції

(процесу або послуг) конкретному стандарту або технічним умовам з опіркою спеціальним знаком або свідоцтвом.

Сертифікат на продукцію засвідчує відповідність якості цієї продукції визначеним вимогам, в умовах ринкових відносин підвищує конкурентоспроможність продукції, є засобом завоювання ринку і захисту споживачів від неякісної продукції.

Спеціальним декретом Кабінету Міністрів України "Про стандартизацію і сертифікацію" передбачається два види сертифікації продукції: обов'язкова і добровільна.

Обов'язкова сертифікація орієнтована на здійснення в Державній системі сертифікації — УкрСЕПРО — перевірки відповідності реальних властивостей продукції, що сертифікується, вимогам, визначеним державними нормативними документами. До обов'язкової сертифікації віднесено потенційно небезпечні та шкідливі продукти, вироби, процеси. У цьому переліку не зазначено програмну продукцію, хоча помилки в ній можуть призвести до небезпечних наслідків як для людей, так і для економіки.

Прикладом небезпечних наслідків можуть служити аварії під час запуску космічних кораблів "Челенджер" (США, 1986) і "Зеніт-2" (СНД, 1998), причиною яких виявилися помилки в програмах керування польотом.

У зв'язку з цим стає зрозумілим, що сертифікація програмної продукції як механізму керування якістю, організації її безпеки і конкурентоспроможності вітчизняних програмних продуктів, захисту користувачів від неякісної продукції конче необхідна. Проте багато організацій, котрі розробляють програмне забезпечення, не планують заходів для забезпечення якості продукції та її сертифікації і пояснюється це такими причинами:

- небажанням піддавати програмне забезпечення сертифікації, бо це потребує додаткових ресурсів;
- нерозумінням з боку замовника програмного забезпечення переваг сертифікованого продукту;
- вадами в організаціях систем забезпечення якості;
- тим, що немає ринку вітчизняної програмної продукції.

Отже, системи забезпечення якості програмних продуктів, нормативно-методичні документи, які визначають найбільш раціональні й ефективні процеси і процедури життєвого циклу, а також системи сертифікації програмного забезпечення спрямовані на вирішення таких завдань:

- 1) створення нормативної бази інженерії якості програмного забезпечення, що відповідає вимогам міжнародних і державних стандартів;
- 2) розробка типових елементів систем забезпечення якості в організаціях, котрі розробляють програмне забезпечення;

3) освоєння й удосконалення сучасних методів оцінювання якості програмного забезпечення;

4) створення нормативно-методичної та інструментальної бази системи сертифікації програмного забезпечення.

## 9.8. Оцінка надійності програмного забезпечення

Надійність — це здатність програмного забезпечення виконувати необхідні функції в заданих умовах функціонування протягом встановленого періоду часу [10—17]. Надійність є одним з основних показників якості й визначається вона після того, як буде проведено тестування програмного забезпечення.

Під час тестування збираються дані про поведінку програмного забезпечення.

Для забезпечення і вирішення завдань надійності на кожному етапі життєвого циклу визначаються чинники, котрі впливають на надійність, а також об'єкти, які підлягають перевірці з метою визначення того рівня надійності, який потребується.

### 9.8.1. Визначення мети і завдань забезпечення надійності на етапах життєвого циклу

Забезпечення надійності програмного забезпечення розглянемо на таких етапах життєвого циклу:

- специфікація вимог;
- проектування, реалізація, випробування;
- супроводження.

Для успішного проведення робіт із забезпечення надійності на цих етапах життєвого циклу необхідно скласти план-графік, у якому відображаються такі дії:

- виділення керованих і некерованих чинників процесу розробки, які впливають на надійність (керовані чинники — рішення про інспекції, обсяги усіх видів ресурсів у тестуванні, некеровані чинники — параметри середовища функціонування, досвід обслуговуючого персоналу програмного забезпечення, обсяг продукту, можливість зміни вхідних вимог тощо);
- вибір необхідних значень керованих чинників для оцінки досягнення цільових вимог до інтенсивності відмов і застосування необхідних обмежень;
- аналіз чинників з метою визначення їхнього впливу на інтенсивність відмов, які виникають у програмному забезпеченні;
- розробка планів тестування вимог і випробування продукту для оцінки надійності програмного забезпечення, в тому числі завдань специфікації вимог, відповідності їх вимогам стандарту на планування, перевірку й атестацію програмного продукту [14—18].

Далі коротко розглянемо завдання забезпечення надійності, розподілене за кожним етапом життєвого циклу.

На етапі специфікації вимог визначаються завдання і зовнішні специфікації основних (цільових) вимог до програмного забезпечення із заданим кількісним метриком для оцінки надійності, виражених у термінах інтенсивності відмов або можливості безвідмовного функціонування програмного забезпечення.

Визначення цих завдань здійснюється шляхом проведення розробником інтерв'ю з користувачем для формування:

- пріоритетів функцій за критерієм важливості їхньої реалізації в програмному забезпеченні;
- сценаріїв подій під час виконання функцій;
- параметрів середовища, режимів та інтенсивності використання функцій і їхніх відмов;
- характеристик моделі (вхідні й вихідні дані) і вхідного простору для кожної функції;
- категорій відмов та їхньої інтенсивності під час виконання функції в заданому сценарії з числом відмов за одиницю календарного часу.

Результатом роботи з користувачем з аналізу вимог є:

- класифікація відмов програмного забезпечення (за ступенем важливості);
- обґрунтування вимог до інтенсивності відмов на основі балансу між надійністю, вартістю і зусиллями, що витрачаються на випуск продукту;
- визначення ставлення замовника до відмов різних ступенів важливості та його готовності платити за зниження інтенсивності відмов за кожною категорією важливості;
- побудова функціональної конфігурації програмного забезпечення з вказівкою частоти використання кожної функції;
- розбивка вхідного простору кожної функції на категорії даних для наступного функціонального тестування програмного забезпечення на етапі випробувань;
- оцінка узгодження календарного часу роботи програмного забезпечення з процесорним.

На етапі проектування визначаються:

- розміри, тобто інформаційна й алгоритмічна складність усіх типів компонент, що проектується;
- категорії дефектів, властиві всім типам компонент програмного забезпечення;
- стратегії функціонального тестування компонентів за принципом “чорної скриньки” за допомогою тестів з метою цілеспрямованого тестування і виявлення дефектів для аналізованої категорії даних програмного забезпечення.

Для окремих компонент програмного забезпечення, різних за функціональним призначенням, проводиться:

- класифікація можливих дефектів і ступеня поширення їх у програмному забезпеченні;
- визначення середовища, режимів й інтенсивності використання компоненти;
- оцінювання ризику використання деякої компоненти у визначеному контексті системи;
- оцінювання впливів відмов компоненти на стійкість всієї системи тощо.

Для досягнення надійного продукту аналізуються:

- варіанти архітектури програмного забезпечення відповідно до поставлених вимог до надійності;
- види технологій аналізу ризику, режимів відмов, дерев помилок та перелік критичних компонентів для забезпечення властивостей відмовостійкості та відновлюваності програмного забезпечення;
- прогнозовані показники розмірів програмного забезпечення, чутливість до помилок і ступінь тестованості, оцінки ризику і складності.

Для одержання надійного програмного забезпечення група якості може вдатися до перерозподілу деяких завдань і до концентрації зусиль на інших завданнях, як наприклад, добір нових повторно використовуваних компонентів, перепроєктування компонентів з низькою надійністю та ін.

На етапі реалізації проектні специфікації трансформуються в тексти мовою програмування і виконується підготовка наборів тестів для автономного й комплексного тестування.

Під час проведення автономного тестування забезпечення надійності полягає в попередженні появи дефектів в окремих компонентах програмного забезпечення і створенні ефективних методів захисту від них. Всі наступні етапи розробки не можуть забезпечити надійність, а лише сприяють підвищенню рівня надійності за рахунок виявлення помилок, що залишилися, за допомогою тестів багатьох категорій.

Забезпечення надійності на цьому етапі досягається за рахунок:

- застосовуваної методології збору й аналізу інформації про аномалії, дефекти і відмови;
- методології виявлення й локалізації дефектів багатьох категорій;
- формування критеріїв завершення тестування, встановлення термінів завершення тестування та вартості всіх ресурсів тестування.

На етапі випробувань створюється план випробувань продукту, за яким проводиться його тестування на відповідність зовнішнім специфікаціям функцій і завданням проекту.

Випробування програмного продукту проводять група якості або користувач у реальному середовищі функціонування чи на випробувальному стенді для імітації функцій компонентів за планами випро-

бувань. Під час підготовки до випробувань вивчається “історія” тестування на стадіях життєвого циклу з метою безпосереднього використання раніше розроблених тестів, а також упорядкування спеціальних тестів випробувань.

На етапі випробувань відповідно до плану забезпечення якості здійснюється:

- оцінювання надійності за результатами системного тестування і польових випробувань за прийнятими для цих завдань відповідними моделями надійності;
- керування зростанням надійності шляхом багаторазового виправлення і регресивного тестування продукту;
- ухвалення рішення про ступінь готовності програмного продукту і про можливість його передачі в експлуатацію.

На етапі супроводження потреба в тестуванні й оцінці надійності продукту може виникнути при виявленні помилок і внесенні змін. З цією метою проводиться:

- протоколювання відмов у ході функціонування програмного забезпечення і вимір надійності функціонуючого програмного забезпечення, а також використання результатів вимірів втрат надійності впродовж експлуатації;
- аналіз частоти і серйозності відмов для визначення порядку усунення відповідних помилок;
- оцінювання впливу функціонування програмного забезпечення на надійність в умовах удосконалених технологій або нових інструментів розробки програмного забезпечення.

### 9.8.2. Моделі та методи оцінки надійності

Моделі для оцінки надійності програмного забезпечення базуються на застосуванні математичного апарату і при допущеннях процесів, що впливають на надійність. На цей час розроблено значну кількість (>100) моделей надійності програмного забезпечення та їхніх модифікацій.

Запропоновано кілька видів класифікації моделей, які ґрунтуються переважно на історії помилок, виявлених у програмах під час тестування їх на етапах життєвого циклу.

Однією з перших класифікацій моделей надійності є класифікація Хетча [11], яка пропонує поділ моделей на прогнозуючі, вимірювальні й оціночні (рис. 9.4).

Прогнозуючі моделі надійності основані на вимірі технічних характеристик створюваної програми: довжина, складність, число циклів і ступінь їхньої вкладеності, кількість помилок на сторінку операторів програми тощо. Наприклад, моделі Мотли — Брукса і Холстеда ґрунтуються на довжині і складності структури програм (кількість гілок, циклів, вкладеність циклів), на кількості і типах

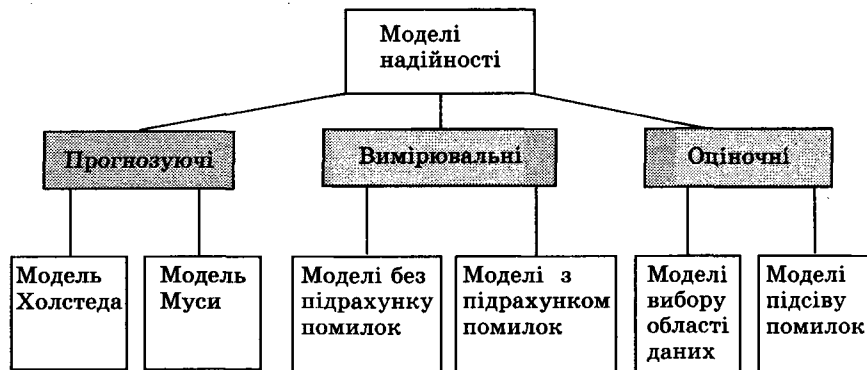


Рис. 9.4. Класифікація моделей надійності

змінних, а також інтерфейсів. У цих моделях довжина програми слугує для прогнозування кількості помилок, наприклад, для 100 операторів програми можна змодельовати інтенсивність відмов.

*Вимірювальні моделі* призначено для виміру надійності програмного забезпечення, яке працює із заданим зовнішнім середовищем, при таких обмеженнях: програмне забезпечення не модифікується під час періоду вимірів властивостей надійності; виявлені помилки не виправляються; вимір надійності виконується для зафіксованої конфігурації.

Типовим прикладом таких моделей є модель Нельсона і Рамамурти — Бастани та інші [11].

*Оцінчні моделі* ґрунтуються на серії тестових прогонів, які проводяться на етапах тестування програмного забезпечення. У тестовому середовищі визначається можливість відмови програми під час її виконання або тестуванні.

Всі наведені типи моделей можуть використовуватися на етапах життєвого циклу. Крім того, результати прогнозуючих моделей можуть використовуватися як вхідні дані для оцінкової моделі. Є моделі (наприклад, модель Муси [11]), які можна розглядати як оцінкові та вимірювальні моделі.

Інший вид класифікації моделей запропонував Гоел [11], за яким моделі надійності базуються на відмовах і розбиваються на чотири класи моделей:

- без підрахунку помилок;
- з підрахунком відмов;
- з підсівом помилок;
- моделі з вибором областей вхідних значень.

*Моделі без підрахунку помилок* оснований на вимірюванні інтервалу часу між відмовами. Вони дозволяють спрогнозувати кількість

помилки, які залишилися в програмі. Після кожної відмови оцінюється надійність і визначається середній час до виникнення такої відмови. До такої моделі належать моделі Джелінські і Моранди, Шика — Волвертона і Литвуда — Вералла [11].

*Моделі з підрахунком відмов* базуються на підрахунку кількості помилок, виявлених на заданих інтервалах часу. Виникнення відмов залежно від часу є стохастичним процесом з беззупинною інтенсивністю, а кількість відмов є випадковою. Виявлені помилки усуваються, і тому кількість помилок за одиницю часу зменшується. До цього класу моделей належать моделі Шумана, Шика — Вулвертона, пуассонівська модель та інші [11].

*Моделі з (так званим) підсівом помилок* оснований на кількості усунутих помилок і підсіві (внесенні) в програму штучних помилок, тип і кількість яких заздалегідь відомі. Потім визначається відношення числа прогнозованих помилок, що залишилися, до числа штучних помилок, що порівнюється з відношенням числа виявлених дійсних помилок до числа виявлених штучних помилок.

Результат порівняння використовується для оцінки надійності і якості програми. Під час внесення змін у програму проводиться повторне тестування й оцінюється надійність. Цей підхід до організації тестування відрізняється громіздкістю і рідко використовується через додатковий обсяг робіт, пов'язаних із добором, виконанням та усуненням штучних помилок.

*Моделі з вибором області* вхідних значень ґрунтуються на генерації множини тестових вибірок із вхідного розподілу й на оцінці надійності, що проводиться за отриманими відмовами на основі тестових вибірок із вхідної області. До цього типу моделей належить модель Нельсона та інші [11].

Є ще один підхід до класифікації моделей надійності, який базується на методичній основі аналітичних моделей і включає статичні й динамічні моделі — моделі зростання надійності.

Нижче розглядаються деякі моделі надійності, котрі забезпечують зростання надійності ПЗ, широко застосовуються на етапі тестування й описують процес виявлення відмов при таких основних припущеннях:

- усі помилки в програмному забезпеченні не залежать одна від одної з погляду локалізації відмов;
- інтенсивність відмов пропорційна поточному числу знайдених помилок у програмному забезпеченні (зменшується при тестуванні програмного забезпечення);
- можливість локалізації відмов залишається постійною;
- локалізовані помилки усуваються до того, як тестування буде продовжено;
- при усуненні помилок нові помилки не вносяться.

Наведемо основні позначення використовуваних формул та термів моделей зростання надійності:

$m$  — число виявлених відмов програмного забезпечення за час тестування;

$X_i$  — інтервали часу між відмовами  $i-1$  та  $i$ , при  $i = 1, \dots, m$ ;

$S_i$  — моменти часу відмов (тривалість тестування до  $i$ -відмови);

$T$  — тривалість тестування програмного забезпечення (час, для якого визначається надійність);

$N$  — оцінка числа помилок у програмному забезпеченні на початку тестування;

$M$  — оцінка числа помилок, які залишилися після тестування;

$MT$  — оцінка середнього часу до чергової відмови;

$E(Tp)$  — оцінка середнього часу до завершення тестування;

$Var(Tp)$  — оцінка дисперсії;

$R(t)$  — функція надійності програмного забезпечення;

$Z_i(t)$  — функція ризику в момент часу  $t$  між  $i-1$  та  $i$ -відмовами;

$c$  — коефіцієнт пропорційності;

$b$  — частота виявлення помилок.

Далі розглядаються моделі зростання надійності.

**Модель Джелінські — Моранди.** У цій моделі [11] використовуються такі вхідні дані:

$m$  — число виявлених відмов за час тестування;

$X_i$  — інтервали часу між відмовами;

$T$  — тривалість тестування.

Функція ризику  $Z_i(t)$  має вигляд:

$$Z_i(t) = c(N - n_{i-1}),$$

де  $i = 1, \dots, m$ ;  $T_{i-1} < t < T_i$ .

Ця функція вважається ступінчатою кусково-постійною функцією з постійним коефіцієнтом пропорційності —  $c$ .

Оцінка параметрів  $c$  та  $N$  провадиться за допомогою системи рівнянь:

$$\sum_{i=1}^m 1/(N - n_{i-1}) - \sum_{i=1}^m cX_i = 0;$$

$$n/c - NT - \sum_{i=1}^m X_i n_{i-1} = 0.$$

При цьому сумарний час тестування обчислюється так:  $T = \sum_{i=1}^m X_i$

Вихідні показники для оцінки надійності щодо зазначеного часу  $T$  включають:

- прогнозоване число помилок, що залишилися,  $M_T = N - m$ ;
- середній час до чергової відмови  $MT_T = 1/(N - m) c$ ;
- середній час до завершення тестування і його дисперсію

$$E(Tp) = \sum_{i=1}^{N-m} (1/ic);$$

$$Var(Tp) = \sum_{i=1}^{N-m} 1/(ic)^2.$$

При цьому функція надійності обчислюється за формулою:

$$R_T(t) = \exp(- (N - m) ct),$$

при  $t > 0$  і числі помилок, виявлених на кожному інтервалі тестування, є рівним одиниці.

**Модель Шика — Волвертона.** Модель [11] використовується, коли інтенсивність відмов пропорційна не лише числу знайдених помилок, а й часу, що минув з моменту останньої відмови. Вхідні дані для цієї моделі аналогічні до розглянутої вище моделі Джелінські — Моранди:

$m$  — число виявлених відмов за час тестування;

$X_i$  — інтервали часу між відмовами;

$T$  — тривалість тестування.

Функції ризику  $Z_i(t)$  у момент часу  $t$  між  $i-1$  та  $i$ -відмовами визначаються так:

$$Z_i(t) = c t (N - n_{i-1}), \text{ де } i = 1, \dots, m; T_{i-1} < t < T_i,$$

$$T = \sum_{i=1}^m X_i.$$

Ця функція є лінійною всередині кожного інтервалу часу між відмовами і знову зростає з меншим кутом нахилу.

Оцінка  $c$  та  $N$  обчислюється за системою рівнянь:

$$\sum_{i=1}^m 1/(N - n_{i-1}) - \sum_{i=1}^m X_i^2 / 2 = 0;$$

$$n/c - \sum_{i=1}^m (N - n_{i-1}) X_i^2 / 2 = 0.$$

До вихідних показників надійності щодо  $T$  належать:

- прогнозоване число помилок, що залишилися,  $M_T = N - m$ ;
- середній час до чергової відмови  $MT_T = (p / (2(N - m) c))^{1/2}$ ;
- середній час до завершення тестування і його дисперсія

$$E(T_p) = \sum_{i=1}^{N-m} (\pi / (2ic))^{1/2};$$

$$Var(T_p) = \sum_{i=1}^{N-m} ((2 - \pi) / 2) / ic,$$

коли число помилок, віддалених на кожному інтервалі тестування після періоду  $T$ , дорівнює одиниці.

Функція надійності обчислюється за формулою:

$$R_T(t) = \exp(- (N - m) ct^2 / 2), \quad t \geq 0.$$

**Модель Гоела — Окумото.** В основі цієї моделі [11] лежить опис процесу виявлення помилок неоднорідним пуассонівським процесом. Її можна розглядати як модель експоненційного зростання. У цій моделі інтенсивність відмов також залежить від часу. Крім того, кількість виявлених помилок трактується як випадкова величина, значення якої залежить від тесту й інших умовних чинників.

Вхідні дані моделі аналогічні до даних попередньої моделі:

$m$  — число виявлених відмов за час тестування;

$X_i$  — інтервали часу між відмовами;

$T$  — тривалість тестування.

Функція середнього числа відмов, виявлених на момент  $t$ , має вигляд:

$$m(t) = N(1 - e^{-bt}),$$

де  $b$  — інтенсивність виявлення відмов і  $q(t) = b$  є показник зростання надійності.

Функція ризику  $\lambda(t)$  від часу роботи до відмови дорівнює

$$\lambda(t) = Nbe^{-bt}, \quad t \geq 0.$$

Остаточна оцінка  $b$  та  $N$  отримується з рівнянь:

$$m/N - 1 + \exp(-bT) = 0;$$

$$m/b - \sum_{i=1}^m t_i - N_m \exp(-bT) = 0.$$

Вихідні показники надійності щодо  $T$  визначають:

1) середнє число помилок, які може бути виявлено в інтервалі  $[0, T]$

$$E(N_T) = N \exp(-bT);$$

2) функцію надійності

$$R_T(t) = \exp(-N(e^{-bt} - e^{-b(t+T)})), \quad t \geq 0.$$

У цій моделі кількість виявлених помилок трактується як випадкове число, яке залежить від тесту і середовища. В інших моделях кількість виявлених помилок розглядається як константа.

У моделях зростання надійності вихідною інформацією для розрахунку надійності є тимчасові інтервали між відмовами тестованої програми, число відмов і час, для якого визначається надійність програми при відмові. На підставі цієї інформації за допомогою моделей визначаються такі показники надійності:

- можливість безвідмовної роботи;
- середній час до чергової відмови;
- число невиявлених відмов (помилки);
- середній час додаткового тестування програми.

У моделі аналізу результатів прогону тестів використовується у своїх розрахунках загальне число експериментів тестування і число відмов. Ця модель визначає лише можливість безвідмовної роботи програми й обирається для випадків, коли попередні моделі не можна використовувати (мало даних, некоректність обчислень). Формула визначення можливості безвідмовної роботи з числа проведених експериментів така:

$$P = 1 - Nex/N,$$

де  $Nex$  — число помилкових експериментів;

$N$  — число всіх проведених експериментів з перевірки роботи системи.

Отже, розглянуто основні широко відомі моделі оцінки надійності, основані на відмовах при тестуванні програм.

## 9.9. Експериментальні підходи до оцінки надійності

Розглянуті вище моделі оцінки надійності в цілому базуються на кількості помилок і відмов, отриманих при тестуванні програмної системи на заданому інтервалі часу. Вони дозволяють також спрогнозувати число помилок, які залишилися в системі. Однак, практично теоретичні моделі використовуються досить рідко через різноманітні обмеження й умови, що накладаються на процес тестування.

Найчастіше використовуваним критерієм оцінки результатів тестування є інтенсивність виявлення помилок та дефектів у програмах на заданому інтервалі часу із застосуванням великого діапазону тестових даних.

При наявності значної кількості помилок та витраченого часу обчислюється надійність, яка на графіку (рис. 9.5) має вигляд кривої.

Інтенсивність усунення помилок майже не впливає на зростання надійності. Проте при застосуванні ефективних методів тестування наробіток на відмову помітно зростає. Відповідно зростає надійність, наближаючись до 1.

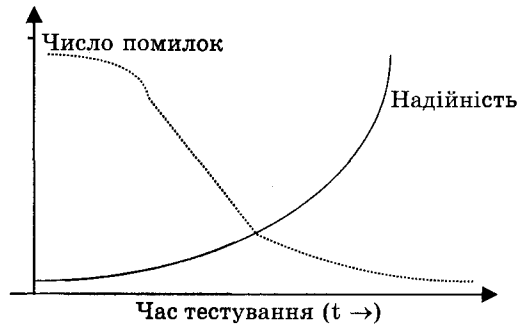


Рис. 9.5. Графік залежності кількості помилок та часу тестування

Експерименти з тестування великих систем показали, що інтенсивність виявлення помилок може коливатися від 0,002 до 0,005 помилок за день [14], тобто в системі може виявитися приблизно одна помилка протягом двох місяців її експлуатації.

На практиці застосовуються інші інтуїтивні методи. Прикладом є метод оцінки числа помилок при тестуванні поточних версій операційної системи OS — 360 фірми IBM (див. гл. 10) за інтуїтивно підбраною формулою:

$$n = 23(m) + 2(k),$$

де  $n$  — загальна кількість змін програми, пов'язаних із виправленням виявлених помилок,

$m$  — число модулів з більше ніж 10 виправленнями в кожному,

$k$  — число модулів з 1 або 2 виправленнями.

Виявилось, що ця формула найповніше відображала надійність роботи версії OS — 360 (370) та її стан на момент передачі замовникові поточної версії.

## 9.10. Методи оцінки рівня якості програмного забезпечення

Відповідно до ДСТУ 2850-94 [5] нижче розглядаються методи оцінки якості програмного забезпечення.

Під **оцінюванням якості** програмного забезпечення розуміють дії, спрямовані на визначення ступеня задоволення програмного забезпечення потребам відповідно до призначення. Метою оцінювання якості є:

- ухвалення рішення про ступінь відповідності досягнутого рівня якості розробленого програмного забезпечення заданому рівню;
- опис ознак та властивостей, реалізованих у програмному забезпеченні;
- атестація й сертифікація програмного продукту.

Під **рівнем якості** програмного забезпечення розуміють відносну характеристику якості, яка базується на порівнянні фактичних значень показників якості оцінюваного програмного забезпечення з базовими значеннями кращих аналогів або із встановленими вимогами на розробку програмного забезпечення.

Номенклатура властивостей і показників якості має багаторівневу ієрархічну структуру (див. рис. 9.2). Перший рівень — це група характеристик (показників) якості, другий — це їхні підгрупи і третій — одиничні (атомарні) показники якості. Кожній властивості відповідають свої показники, які мають назву, що збігається з назвою властивості (наприклад, показник надійності).

Для оцінки рівня якості програмного забезпечення можуть застосовуватися диференційований, комплексний або змішаний методи.

**Диференційованим** називається метод оцінки рівня якості продукції, оснований на використанні окремих показників (атрибутів) якості. При цьому визначають, чи досягнуто в цілому рівня базових значень програмного забезпечення, а також чи досягнуто або чи не досягнуто показників.

При диференційованому методі відносні значення окремих показників обчислюються за формулами:

$$Q_i = \frac{P_i}{P_{ib}}, \quad (1)$$

$$Q_i = \frac{P_{ib}}{P_i}, \quad i = 1, 2, \dots, N, \quad (2)$$

де  $P_i$  — значення  $i$ -го показника якості оцінюваного програмного забезпечення;

$P_{ib}$  — базове значення  $i$ -го показника;

$N$  — кількість показників якості.

Формула (1) використовується тоді, коли більше за розміром значення показника поліпшує ситуацію. В іншому разі використовується формула (2).

При використанні диференційованого методу рівень якості оцінюваної програмної продукції вважається вищим або таким, що дорівнює рівню базових значень, якщо всі значення відносних показників більші або дорівнюють одиниці. В іншому разі рівень якості оцінюваної програмної продукції є нижчим за рівень базових значень. Якщо частини показників  $Q_i > 1$ , а для іншої частини значень  $Q_i < 1$ , то слід застосовувати комплексний або змішаний методи оцінки.

Рівень якості оцінюваної програмної продукції, для якої суттєво важливе значення кожного показника, вважається нижчим від базового, якщо хоча б один із показників менший за одиницю.

*Комплексний метод* оцінки рівня якості програмної продукції оснований на використанні узагальненого показника, який є функцією від кількох окремих та групових показників і коефіцієнтів їхньої вагомості.

Встановлення коефіцієнтів вагомості показника якості, як правило, провадять експертним шляхом. При цьому широко використовують шкали:

- 5 — надто важливо, щоб даний показник мав високе значення;
- 4 — важливо, щоб даний показник мав високе значення;
- 3 — добре б мати високе значення даного показника;
- 2 — показник має середній вплив;
- 1 — за низьких значень даного показника відчутних втрат немає.

Узагальнене значення коефіцієнта вагомості  $m$  (параметр вагомості  $M$ ) обчислюється за формулою:

$$m_i = \frac{M_i}{\sum_{i=1}^N M_i}, \quad (3)$$

де  $M_i$  — значення  $i$ -го коефіцієнта вагомості в згаданих (або інших) шкалах.

Значення параметрів (коефіцієнтів) вагомості визначаються під час складання вимог технічного завдання на розроблюваний програмний продукт. Якщо до технічного завдання додається план забезпечення якості, то значення параметрів зазначають у цьому плані. Встановлені значення параметрів вагомості переглядаються тільки у разі корекції цих документів.

Якщо  $\sum_{i=1}^N M_i = 1$ , то коефіцієнти вагомості називають параметрами вагомості.

Узагальнений показник якості може бути передано середнім зваженим (арифметичним чи геометричним) або іншим показником якості.

Середній зважений арифметичний показник обчислюється за формулою:

$$U = \sum_{i=1}^N Q_i M_i, \quad (4)$$

де  $M_i$  — параметр вагомості  $i$ -го показника, що входить до узагальненого показника,  $i = 1, 2, \dots, N$ ;

$N$  — кількість показників, які становлять середній зважений показник.

Якщо до моменту випробувань й оцінювання рівня якості програмного забезпечення значення параметрів вагомості показника якості, що використовуються, не було встановлено, то ці значення встановлюються спеціалістами, які проводять оцінку якості. Під час повторних випробувань, як правило, мають використовуватися ті самі показники, які встановлено раніше.

При багаторівневій ієрархічній структурі номенклатури показників якості розрахунок рівня якості програмного забезпечення провадиться знизу (від одиничних показників) догори (до отримання узагальненого показника якості).

*Змішаний метод* оцінки рівня якості ґрунтується на сумісному застосуванні одиничних та комплексних (групових) показників. При цьому частина одиничних показників об'єднуються в групи. Після такого об'єднання обчислюються відносні значення групових показників за формулами (1), (2) і (3).

Порівняння рівня якості оцінюваного програмного забезпечення з базовими показниками якості провадяться, як за диференційованим методом.

Змішаний метод застосовується у тому разі, коли:

- сукупність одиничних показників якості є досить широкою й утруднює отримання узагальнених висновків;
- узагальнений показник якості в комплексному методі не дозволяє отримати висновки про відповідність значень деяких суттєвих групових показників потрібному рівню.

## Контрольні запитання і завдання

1. Наведіть визначення якості програмного забезпечення.
2. Назвіть основні аспекти і рівні моделі якості програмного забезпечення.
3. Наведіть характеристики якості й визначте їхній зміст.
4. Які методи використовуються для визначення показників якості?
5. Які метрики програмного продукту та їхні складові Ви знаєте?
6. Які є стандарти якості програмного забезпечення?
7. Назвіть основну мету і завдання системи керування якістю.
8. У чому суть інженерії якості?
9. Визначте термін *надійність програмного забезпечення* і сформулюйте завдання її забезпечення на етапах життєвого циклу.
10. Які моделі оцінки надійності є і на яких припущеннях вони базуються?

## Література до глави 9

1. NASA — STD — 2201/ Software Assurance Standard, 1993.
2. ANCI/IEEE 730-1. IEEE Standard for Software Quality Analysis Plans, 1989.
3. ISO/IEC 9126. Information Technology. — Software Quality Characteristics and metrics, 1997.

4. ДСТУ 2844—1994. Програмні засоби ЕОМ. Забезпечення якості. Терміни та визначення.

5. ДСТУ 2850—1994. Програмні засоби ЕОМ. Забезпечення якості. Показники та методи оцінювання якості програмного забезпечення.

6. ДСТУ 3230—1995. Управління якістю та забезпечення якості. Терміни та визначення.

7. ISO 14598. Information Technology — Software product evolution — Part 1: General overview, 1996.

8. Meyer B. The role of Object-Oriented Metrics // Computer. — 1998. — № 11. — P. 123—125.

9. Haag S., Raja H.K., Sekade L.L. Quality Function Deployment. Usage in Software Development // Comm. of the ACM. — 1998. — V. 39. — № 1.

10. Кулаков А.Ф. Оценка качества программ ЭВМ. — К.: Техніка, 1984. — 167 с.

11. Мороз Г.Б., Лаврищева Е.М. Модели роста надежности программного обеспечения. — К.: Препринт 92—38, 1992. — 23 с.

12. Jacobson J. Object-oriented Software Engineering. — Revised Printing. — Addison-Wesley, 1995. — 528 p.

13. Pfleeger S.L. Software Engineering. Theory and Practice. — Prentice Hall, 1998. — 576 p.

14. Лунаев В.В. Надежность программных средств. — М.: СИНТЕГ, 1998. — 231 с.

15. Лунаев В.В. Тестирование программ. — М.: Радио и связь, 1986. — 296 с.

16. Майерс Г. Надежность программного обеспечения. — М.: Мир, 1980. — 360 с.

17. Гласс Г. Руководство по надежному программированию. — М.: Финансы и статистика, 1982. — 256 с.

18. Тейер Т., Липов М., Нельсон Э. Надежность программного обеспечения. — М.: Мир, 1981. — 325 с.

19. IEEE Software. Measurement. — 1997 — March/April.

20. Андон Ф.И., Сулов В.Ю., Коротун Т.М., Коваль Г.И. Парадигма качества программного обеспечения // Проблемы программирования, 1999. — Вып. 2. — С. 51—62.

21. Dunham J.R. Verification and Validation in next decade // IEEE Software — 1989. — May. — P. 47—53.

## Глава 10. КЕРУВАННЯ КОНФІГУРАЦІЯМИ ПРОГРАМНИХ СИСТЕМ

### 10.1. Зміст процесу керування конфігурацією

Під *конфігурацією системи* розуміють склад усієї системи або окремих її частин, зв'язків між ними, зумовлений основними технічними і функціональними характеристиками системи та розв'язуваними задачами опрацювання даних [1—4].

Конфігурацією системи можна також вважати сукупність спеціальних компонентів апаратного, програмно-апаратного і програмного забезпечення, об'єднаних у єдине ціле зі спеціалізованими функціями та процедурами підтримки функціонування й виконання конкретних завдань. Чим більша програмна система, тим більше компонентів піддаються змінам, пов'язаним зі знайденими помилками, уточненням або доповненням нових функцій.

**Керування конфігурацією** програмного забезпечення (**КК**) — це порядок визначення конфігурації системи у певні моменти часу її створення, систематичного контролю за внесенням змін у систему, а також підтримки її цілісності й трасованості протягом усього життєвого циклу.

З формального погляду керування конфігурацією полягає в дисциплінованому застосуванні технічного або адміністративного керування і методів нагляду за визначеними і документованими функціональними та фізичними характеристиками окремих складових системи, а також методів керування змінами цих характеристик, у підготовці звітів за виконаними змінами і процедурами їхньої перевірки на відповідність поставленим вимогам.

Концепція керування конфігурацією належить до програмного й апаратного забезпечення. Вона пов'язана з питаннями ідентифікації конфігурації, визначенням статусу й аудиту цієї конфігурації, а також з керуванням розробкою програмної продукції.

Взаємозв'язок цих видів діяльності з керування конфігурацією наведено на рис. 10.1.

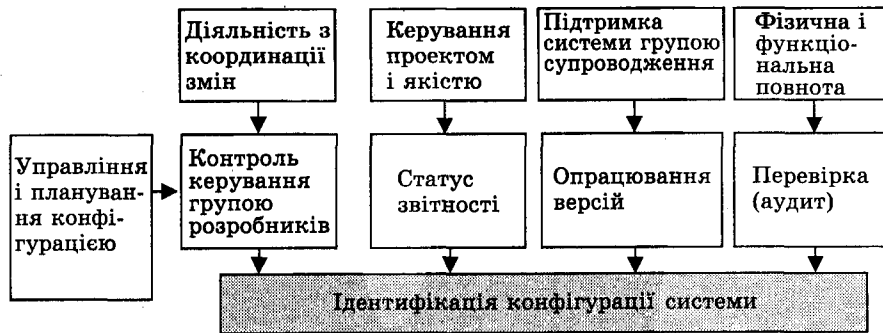


Рис. 10.1. Види діяльності системи керування конфігурацією

Мета й завдання керування конфігурацією полягають у такому:

- визначення необхідності, координація і контроль змін;
- керування проектом і внесенням змін у систему;
- підготовка і контроль версій системи;
- супроводження версії програмної системи та її аудит;
- визначення фізичної та функціональної повноти системи і витрат ресурсів.

Результатом виконання цих завдань є звітність про проведені зміни, визначення версії системи і документації на неї, документ про контроль і передачу версії користувачам.

Для того, щоб провести системну роботу з виконання необхідних змін системи, доцільно створити спеціальну службу, обов'язками якої є:

- визначення обмежень, планування й організація керування конфігурацією;
- визначення інструментарію для керування конфігурацією та внесення змін;
- організація перевірок (аудит) станів частин системи;
- внесення змін, збір даних, облік, перевірка системи після внесення змін тощо.

Структура такої служби залежить від складності й етапів розвитку проекту, від структури організації-розробника та її взаємодії із замовником. Чіткість організації цієї служби і її фактичне місце в структурі організації визначають ефективність контролю й керування конфігурацією системи.

Найбільшу роль керування конфігурацією системи відіграє на кінцевому етапі супроводження.

## 10.2. Керування конфігурацією як процес

Керування конфігурацією програмного забезпечення — це процес підтримки життєвого циклу програмного продукту, що визначає успіх проекту і лінію керування, тобто діяльність з розгортання проекту і гарантії одержання якісного продукту для кінцевих замовників та користувачів. Для досягнення успіху керування конфігурацією потрібно його планувати. Це приводить до розуміння організаційного контексту і до розгляду необхідних обмежень при здійсненні керування конфігурацією.

**Організаційний аспект керування.** Для того, щоб спланувати процес керування в проекті, створюється спеціальна організаційна структура (служба) і система відношень між її складовими. Взаємодія між членами служби, розробниками, користувачами необхідна під час виконання всіх видів діяльності в процесі життєвого циклу розробки системи. На рис. 10.2 подано можливу організаційну структуру і взаємозв'язок ролей служби з керування конфігурацією системи.

Спеціалісти, відповідальні за процес проектування й кодування, складають основу організаційної структури підприємства, а відповідальність за виконання основних завдань керування модифікацією системи покладається на служби керування конфігурацією, а також на головного програміста проекту організації, який займається розробкою програмного забезпечення.

Відповідно до наведеної схеми на рис. 10.2 основну функцію й ухвалення рішень про внесення змін у конфігурацію системи виконує служба керування конфігурацією. Її робота може полягати в такому. Замовником пропозицій на внесення змін може бути як розробник, так і споживач. Пропозиції на зміни в системі надходять до групи збору змін, яка робить оцінку вартості відповідних робіт щодо змін системи і виробляє пропозиції для керування конфігурацією.

Служба конфігурації проводить аналіз та визначає доцільність внесення змін з урахуванням проведеної оцінки вартості і передає затверджений перелік змін до їхньої реалізації шляхом коригування системи та її випробування. Результати випробувань передаються службі, котра затверджує зміни, якщо модифікацію системи відображено в документації і подано в новій версії.

Далі необхідно здійснити тиражування документації і передачу системи замовникам. Група обліку версій системи посилає всім споживачам повідомлення про припинення супроводження попередньої версії та появу нової системи.

У зв'язку з тим, що до утворюваної великої системи входять елементи апаратної і програмної частин, діяльність з керування конфігурацією системи з цими двома частинами ведеться паралельно.

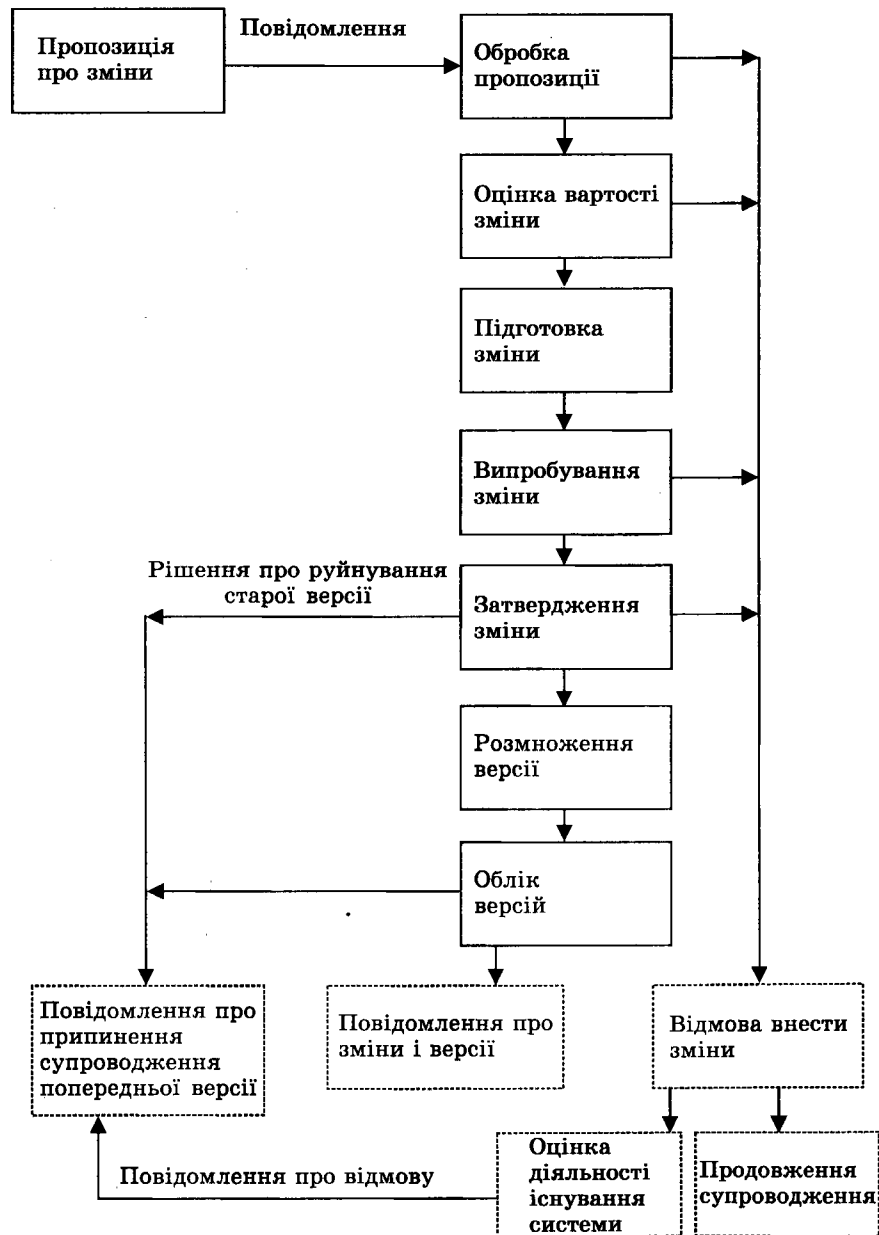


Рис. 10.2. Організаційна структура і взаємозв'язок у службі керування

На етапах життєвого циклу робляться відповідні записи за результатами проектування, які можуть послужити гарантією досягнення необхідної якості продукту. Отже, є тісний зв'язок між видами діяльності розробників, спрямованих на побудову моделі життєвого циклу системи, планів і завдань розробки, а також на визначення інструментаріїв ведення розробки. При цьому є інструментарії, котрі підтримують як ведення розробки продукту, так і вирішення завдань керування конфігурацією.

**Обмеження на створення продукту.** Обмеження на створення системи виявляються з різних джерел (специфікації вимог, описів, звітів та ін.). На їхнє формування впливають будь-які організаційні заходи, наприклад контакти із замовником, аудитором, що визначають основні кроки і пункти в контролі конфігурації. Важливу роль відіграють інструментальні засоби, за допомогою яких встановлюються обмеження на розробку, а також державні стандарти або стандарти інших організацій, які пройшли відповідну апробацію.

**Планування керуванням.** Планування залежить від типу проекту, організаційних заходів, обмежень галузі застосування (розміру, критичності) і загальних рекомендацій щодо керівництва конфігурацією.

До основних видів діяльності з планування конфігурацією системи (рис. 10.3) належать ідентифікація конфігурації, визначення статусу й аудиту конфігурації, керування змінами та їхньою підтримкою.

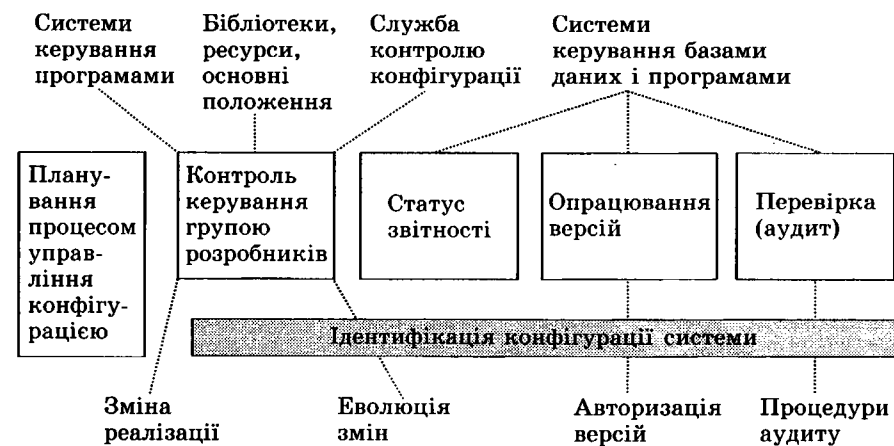


Рис. 10.3. Характеристика інструментів і процедур діяльності керування конфігурацією

Процес планування полягає в упорядкуванні планів, виборі інструментарію, дотриманні вимог проекту, керуванні субпідрядниками й інтерфейсами компонентів та користувачів тощо.

До елементів підтримки діяльності з керування конфігурацією належать:

- система керування вихідним кодом програм, що забезпечує переведення й об'єднання елементів із бібліотеки, та перевірку конфігурації системи;
- базові системні бібліотеки і ресурси, що забезпечують виконання завдань керування конфігурацією відповідно до графіка проекту розробки й упорядкування проміжного звіту;
- спеціальна служба з проведення контролю і керування конфігурацією системи;
- системи керування базами даних та програмами, що сприяють накопиченню даних з ведення проекту, веденню вироблених змін і версій системи.

За допомогою перелічених засобів проводяться необхідні зміни, авторизація версій конфігурації системи, звітність з усіх видів змін і аудит.

Основними завданнями планування керування конфігурацією є:

- фіксація виконавцями окремих завдань на зміни відповідно до пріоритетів складових мети керування проектом у рамках організації й узятих обов'язків з виконання проекту;

- визначення людино-часових та інструментальних ресурсів, необхідних витрат для встановлення правильної послідовності змін, зв'язків між ними, а також розвиток плану модернізації системи або основних його напрямів. Усі ці вимоги слід заздалегідь обговорити і відзначити в потрібних місцях плану з вказівкою відповідальних осіб;

- вибір відповідного інструментарію для підтримки керування конфігурацією системи з кількома типами компонентів та цільовими завданнями. Наприклад, інструментарій для роботи з кодами компонент оперує з бібліотекою цих компонент і координує діяльність багатьох користувачів цієї бібліотеки, а також використовує засоби ведення документації на етапах процесу керування конфігурацією, внесення змін у програмний продукт і керування базами даних;

- встановлення зв'язків із замовником для визначення його вимог на продукт, здійснення контролю і керування виконанням продукту, а також для оцінки широко використовуваних і дорогих продуктів (наприклад компіляторів);

- визначення чіткої послідовності кроків процесу керування конфігурацією, їхніх зв'язків та впливів один на одного, переходу від одного етапу до іншого з урахуванням проведених змін на визначених кроках процесу, контроль внесення змін в інші частини конфігурації системи.

Результати планування доцільно відобразити в спеціальному документі — плані керування конфігурацією у формі описів етапів керування конфігурацією проекту, а також як внесення змін протягом усього періоду життєвого циклу системи. До складових плану керування належать:

- визначення цілей, розмірів і меж проекту, а також базової термінології системи керування конфігурацією;
- розподіл обов'язків, що включає організацію, обов'язки, застосовувані заходи, вказівки і процедури;
- визначення завдань окремих груп з керування конфігурацією;
- плани координаційної діяльності з іншими проектами і групами;
- ресурси, які включають інструментальні, фізичні та людські ресурси; засоби підтримки й забезпечення планів управління конфігурацією.

Контроль керування конфігурацією програмних частин проекту проводять ініціатори проекту на кількох етапах його ведення.

**Метрики керування конфігурацією.** Програмні бібліотеки й різноманітні інструменти підтримки процесів одержання необхідної інформації про різноманітні поточні характеристики ведення проекту (наприклад тимчасові), а також виявлення, аналіз і перевірка отриманих характеристик оцінюються відповідно до норм та мір, встановлених замовником. Менеджмент керується визначеними числовими показниками оцінки процесу керування за допомогою системи показників. Їм відповідають метрики, котрі використовуються для виміру необхідних характеристик. Як метрики використовуються міри, що визначають час роботи компоненти, число виявлених помилок, відмов інструментів, порушень стандартів або введеної термінології тощо.

### 10.3. Ідентифікація конфігурації

Конфігураційна ідентифікація включає методи, за допомогою яких встановлюються поточний склад системи, конкретний зміст виконаних до визначеного моменту часу змін у кожному примірнику, а також відповідність між системою і документацією.

З метою побудови схеми ідентифікації основних частин системи і кількох версій визначаються інструменти й технології для опису та керування основними пунктами конфігурації. Діяльність з ідентифікації системи полягає у виконанні таких завдань у процесі управління конфігурацією:

- визначення основних питань керування ідентифікацією, зокрема, розробка стратегії виявлення змін, оцінки елементів конфігурації і системи відношень між ними;
- представлення складових частин і всієї конфігурації програмної системи як набору функціональних і фізичних характеристик програмного забезпечення, що мають відображатися в документації системи;
- виділення елементів конфігурації або окремих сутностей у процесі керування конфігурацією. Потенційно елементом (або відзначеною частиною) конфігурації можуть бути плани, специфікації, матеріали тестів, програмні інструменти, джерела коду і виконувані коди, програмні бібліотеки, дані і словники даних, документація на про-

грамне забезпечення, а також операції звернень до програмного продукту. На підставі цих даних встановлюється правильне співвідношення між кількістю виконуваних завдань проекту і кількістю пунктів конфігурації;

- визначення відношень між елементами конфігурації та їхніми частинами, що залежать від інших видів діяльності процесу управління конфігурацією (наприклад, побудова продукту або аналіз впливів запропонованих змін), оцінка їх у спеціальних пунктах конфігурації і проведення змін;

- ведення версії програмного забезпечення (або його елементів), документування її та оцінка в пунктах конфігурації. Ревізія й випробування версії, заміна старої версії на нову версію, формування нового варіанта конфігурації із заміщенням попередньої версії;

- вибір програмних елементів, які утворюють основне ядро конфігурації, їхнє формальне позначення; фіксація змін у визначені моменти часу життєвого циклу за допомогою спеціальних типів процедур, орієнтованих на поточне удосконалення проекту. До ядра належать: функціональна лінія, спрямована на відстеження зв'язків відповідно до вимог проекту, розподілена за специфікаціями вимог та інтерфейсами частин системи; лінія розвитку, орієнтована на еволюцію конфігурації програмної частини проекту в задані моменти часу життєвого циклу;

- керування змінами елементів конфігурації в заданий час життєвого циклу. При цьому лінії об'єднуються в конкретну лінію, а її точки стають точками життєвого циклу для проведення зміни.

**Бібліотека програмних компонентів** — це керована сукупність (збірка) елементів програмної системи та відповідної документації, пов'язаної з розробкою, використанням і підтримкою програмного забезпечення. Типи бібліотек мають рівні й етапи розвитку програмних продуктів. Наприклад, робоча бібліотека підтримує етап кодування, а бібліотека головного програміста — остаточне формування кінцевого продукту.

Бібліотеку, як інструментарій керування конфігурацією, призначено для підтримки ведення і керування конфігурацією, а робочу бібліотеку — для керування кодами програмної системи в процесі керування конфігурацією.

## 10.4. Організація процесу змін

В основі керування конфігурацією системи лежать процеси керування змінами, які відбуваються під час життєвого циклу продукту. Такий процес включає визначення типів змін, організацію проведення цих змін і формування концепції допустимих відхилень та відмов від вимог у проекті.

За керування конфігурацією відповідає спеціальна служба, яку очолює керівник проекту та модернізації системи. Служба конфігура-

ційного керування здійснює аналіз і контроль проведення змін конфігурації конкретної системи. У свою чергу, замовник оцінює результати цієї служби за аналізом пропозицій на внесення змін і дає дозвіл на проведення найважливіших змін у системі, що впливають на її технічні характеристики або вартість.

Керування конфігурацією дозволяє організовувати і систематично враховувати, а також контролювати внесення обґрунтованих та санкціонованих змін на всіх етапах життєвого циклу. Для вирішення цих завдань проводиться ідентифікація, контроль та облік змін. Розглянемо задані дії докладніше.

**1. Затвердження (реєстрація) змін.** Перший крок керування змінами — це визначення мети проведення змін. Процес визначення необхідних змін підтримується формальними процедурами представлення й запису необхідних змін, оцінки впливу запропонованих змін на вартість, ухвалення рішень про надання згоди на зміни або відмову від них.

Вимоги на зміни на певному стані програмної конфігурації може бути висунуто будь-яким співробітником, задіяним на цьому етапі, й у будь-якій точці життєвого циклу. Технічна оцінка й контроль змін проводяться з метою визначення якості виконаних змін.

Контроль конфігурації забезпечують методи, які дозволяють проводити систематичну оцінку запропонованих змін системи і координацію реалізації їх з урахуванням ефективності та витрат на виконання змін. Затверджені й реалізовані зміни підлягають обліку й реєстрації для забезпечення адекватності продукту і супровідної документації на цей продукт.

**2. Дошка керування конфігурацією.** На ній відображається інформація про зміни відповідального за план і рівень змін на заданому етапі життєвого циклу. У невеликому проекті контрольні повноваження перебувають у керівника або адміністратора дошки.

Повноваження можна делегувати іншим членам групи, які, залежно від критеріїв і природи змін (від таких, як бюджет або графік), можуть оцінити якість змін на етапах життєвого циклу. Композиція використовуваної кількості дощок для заданого варіанта системи залежить від цих критеріїв. Дошка слугує для розробника програмного забезпечення засобом контролю за конфігурацією програмного забезпечення.

**3. Інструментарій керування змінами.** Такий інструментарій призначено для підтримки паперових форм та автоматизованого документування проведених змін. Він може бути електронним засобом, що фіксує причини проведення змін, а також визначає місце збереження описів змін і відповідні їм звітні форми. Фактично засоби керування змінами спеціально готуються для локальних процесів. Об'єднання локальних процесів та інструментів проводиться в середовищі розробки програмного забезпечення.

**4. Реалізація змін.** Запити на внесення змін реалізуються відповідно до визначених процедур розробки програмного забезпечення. Оскільки множини необхідних змін можна провести одночасно, передбачається засіб забезпечення збереження й трасування змін у побудові нових версій. Після етапу проведення змін системи піддаються детальному аудиту.

Нижче наведено схему процесу контролю змін (рис. 10.4), що забезпечує мінімальну оцінку ризику керування конфігурацією.

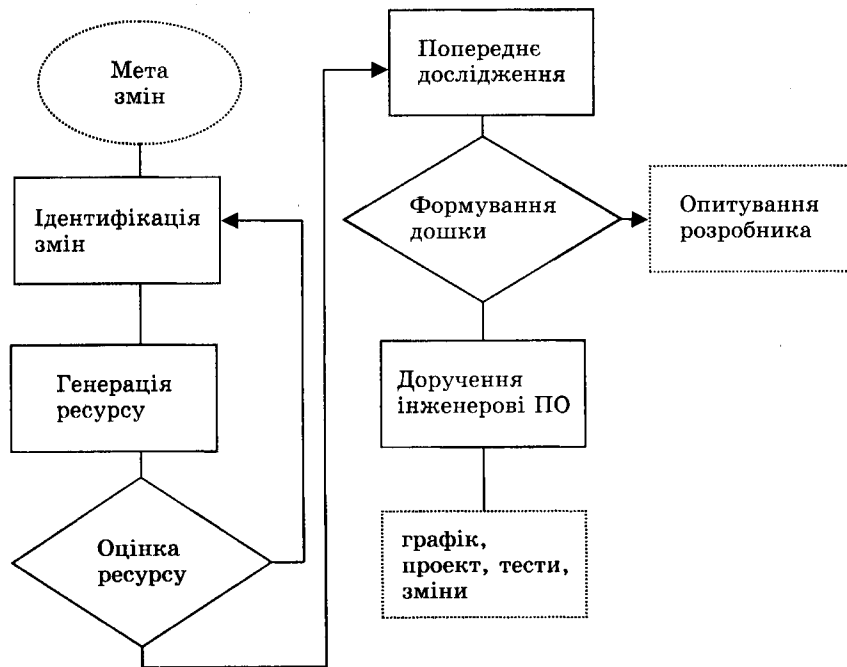


Рис. 10.4. Процес контролю змін

**5. Інструментарій контролю за змінами.** Такий інструментарій призначено для підтримки керування створенням версій програмного продукту і можливістю роботи з репозиторієм вихідних кодів. Як мінімум, такі інструменти забезпечують перевірку вихідного коду й контролю версій. Використанням цих інструментів може займатися незалежна група керування конфігурацією. Подібні інструменти появились як інтеграційні частини продуктів фірм “Rational’s Clear Case”, “Source Safe of Microsoft” і версії контролю конфігурацій у системі UNIX.

**6. Відхилення і відмови.** Обмеження на розробку програмної продукції та описи їх, зроблені під час етапів розробки, можуть породити

відхилення від поставлених вимог на розробку. Під відхиленням розуміють дозволений або недозволений відхід від установлених вимог на розробку елементів конфігурації. Відмова — це рішення про невикористання деякої частини (елемента) конфігурації або можливість змінити елемент системи засобом, що не збігається із запропонованим. У такому разі треба одержати формальний дозвіл на відхилення або відмову від запропонованих вимог до розробки.

## 10.5. Конфігураційний облік

Конфігураційний облік — це методи реєстрації і накопичення звітів про всі реалізовані та відхилені зміни деякого вхідного варіанта системи. Такі звіти в наочній формі відбивають поточний стан системи, історію розвитку і внесення змін. Вони використовуються для запису або перезапису поточної інформації, необхідної для керування конфігурацією програмного забезпечення.

**Інформація про конфігураційний облік.** Інформація про конфігураційний облік визначається діяльністю кількох категорій керівників, котрі ухвалюють рішення про доцільність змін, керуючись накопиченими звітами на етапах життєвого циклу та з урахуванням обмежень бюджетів і термінів. Під час аналізу і вибору змін важливе значення має точне урахування ступеня впливу змін на всі компоненти системи і на показники якості.

Під час ухвалення рішень про внесення змін використовуються відомості про ідентифікацію конфігурації і про всі поточні зміни статусу, а також про прогнозовані відхилення та відмови. Отже, інформація про конфігураційний облік включає затверджену ідентифікацію, діючу реалізацію статусу змін, а також прогнозовані вище відхилення і можливі відмови.

**Засіб конфігураційного обліку.** Інструмент для конфігураційного обліку включає засоби автоматичного збору окремих наборів даних про поставлені завдання побудови конфігурації системи із застосуванням реляційної й об’єктно-орієнтованої системи керування базами даних. Результатом конфігураційного обліку є звіт.

Звітна інформація про конфігураційний облік програмного забезпечення використовується надалі організаціями різних рівнів (робочою групою, керуючою групою тощо) для підтримки варіантів конфігурації. Ця інформація відображається також на рекламному рівні.

Доповненням до звітної інформації з поточного конфігураційного обліку, включеної до бази даних, є різноманітні виміри метричних характеристик за даними про процес розробки і менеджменту. За метрики, наприклад, може використовуватися число проведених змін або часу, витраченого на проведення кожної зміни тощо.

## 10.6. Аудит конфігурації

Аудит конфігурації програмного забезпечення — це перевірка середовища конфігурації на задоволення функціональних та фізичних характеристик системи. Неформальний аудит може проводитися в ключових точках життєвого циклу. Під час укладення договорів розглядається здебільшого формальний аудит видів конфігурації, а також процесів програмного забезпечення:

- аудит функціональної конфігурації програмного забезпечення полягає в перевірці відповідності виготовленої програмної продукції визначеним раніше специфікаціям вимог. Результат подібних верифікацій та сертифікацій є ключовим для аудиту програмного забезпечення;
- аудит фізичної конфігурації полягає у визначенні наявності необхідної документації, що відбиває процес розробки системи;
- аудит процесу розробки програмного забезпечення полягає у визначенні поточного статусу основних елементів конфігурації на даному етапі життєвого циклу й ухваленні рішення про перехід до наступного етапу процесу реалізації програмного продукту.

Правильна комбінація наведених видів аудиту є основою для встановлення основної конфігурації.

## 10.7. Керування версіями та передача версій замовникам

Програмний продукт після внесення змін стає черговою версією, що поширюється виготовлювачем як варіант програмного продукту. Версія використовується для передачі або продажу замовникам елементів конфігурації програмного забезпечення. Коли основна версія програмного продукту вже готова до передачі одержувачеві для використання, як правило, виникає необхідність у підготовці цієї версії з урахуванням можливостей платформ і властивостей наявного середовища одержувача. У зв'язку з цим може виникнути необхідність створення нової версії та пакету матеріалів для її передачі. Керування версіями полягає у виконанні дій, які описуються нижче.

**Збіркова побудова програмного забезпечення.** Збіркова побудова програмного забезпечення — це діяльність з композиції коректної й остаточної версії програмного продукту з елементів конфігурації, накопичених під час реалізації продукту на етапах життєвого циклу, та виконання цієї версії. Виконувальність коду залежить від системи апаратного забезпечення й інструментаріїв (наприклад компіляторів), за допомогою яких будується програмне забезпечення. Інструментарії дозволяють у разі потреби повторно видати точну копію частини програмної конфігурації. І крім того, вони підтримують і процеси керування конфігурацією.

**Інструментарій побудови версії.** Такий інструментарій необхідний для вибору коректної версії програмного продукту або окремих його частин, а також для визначення можливостей середовища та засобів автоматизації процесу побудови кінцевого продукту з кількох варіантів версій з коректною конфігурацією програмного забезпечення та даних. Подібні інструменти мають ступені складності, потребують вивчення спеціальних мов графічного представлення і висококваліфікованих спеціалістів.

**Керування версіями.** Це вид діяльності, спрямованої на створення сукупності елементів, які підтримують подальше використання продукту, як, наприклад, виконувані коди, документація з описом застосування продукту і даних про конфігурацію. Такі відомості необхідні користувачам для подальшої експлуатації версій програмного продукту. Замовникові передаються також інструменти керування версіями програмної продукції для підтримки варіантів застосування набору елементів конфігурації або конкретного випадку їхнього використання (для конкретного замовника). Водночас можливим користувачам передається інформація про варіанти цільових платформ і програмного забезпечення.

**Приклади.** Багатий досвід створення версій здобула широко відома у світі ОС — 360 (1965—1978 рр.) фірми IBM. З моменту першого її випуску за 12 років експлуатації вона мала 21 версію. Це пов'язано, насамперед, з постійним поетапним додаванням нових функціональних можливостей і внесенням змін у попередню версію у зв'язку із знайденими помилками або неповнотою оголошених у версії реалізованих функцій. Над розвитком додаткових можливостей цієї ОС і внесенням змін у попередню версію працював постійний колектив фірми.

На статистику змін операційної системи впливали численні користувачі, котрі в процесі експлуатації чергової версії знаходили різноманітні помилки й інформували розробників. Трудомісткість розробки чергової версії ОС вважалася пропорційною інтервалу часу між реєстраціями чергових версій, коли час приймався за одиницю виміру складності створення нової версії.

Як міра трудомісткості супроводження й створення чергової версії використовувалося число модулів (модулі мали обмежені розміри і стандартизований опис), котрі піддаються змінам і доповненням. Крім того, оцінювалася інтенсивність робіт із створення версії, яка вимірювалася числом змінюваних модулів за одиницю часу. Після 12 років постійних змін остання, 21 версія, стабілізувалася, і претензій з боку користувачів майже не було.

Метричний аналіз процесу розвитку ОС — 360 дозволив встановити, що обсяг середнього приросту системи на кожну версію відповідав приблизно 200 модулям. При цьому загальний обсяг збільшився від 1 тис. модулів у перших версіях до 5 тис. модулів в останніх. При

цьому, коли рівень приросту складності був завеликим, для усунення помилок або додаткових коригувань іноді створювалися проміжні версії з меншим числом змін.

Унаслідок цього з'явилося поняття *критична маса*, яке відповідало поняттю *критична складність частини системи, що модифікується*. Якщо при модернізації і випуску чергової версії системи обсяг доробок перевищує “критичний”, то існує велика можливість погіршення характеристик системи або необхідність введення проміжної версії для проведення деяких змін.

Характерно, що “критичний” обсяг доробок ОС — 360 — близько 200 модулів — залишався (більше 12 років) постійним, незважаючи на зростання кваліфікації колективу, удосконалення технічних і програмних засобів, а також на інші чинники. При цьому в перших версіях обсяг доробок становив 20 % модулів, а в останніх знизився до 5 %.

## 10.8. Управління документацією та супроводженням

Виготовлена конфігурація (версія) програмної системи передається замовникові для її експлуатації та супроводження (див. главу 8).

Разом з версією замовникові передається експлуатаційна документація на систему в електронному варіанті, яка включає такі головні інструкції:

- для адміністратора та операторів, котрі проводять інсталяції системи й управління режимами вирішення функціональних завдань;
- для операторів-користувачів, які використовують систему відповідно до її призначення;
- для навчального й довідкового використання при освоєнні системи за допомогою Help.

Серед документації можуть бути методики, орієнтовані на перенесення системи до другого середовища чи іншої платформи.

Метою супроводження є виявлення й усунення знайдених у системі помилок, аналіз стану системи, коригування документації та її тиражування, контроль версії, актуалізація і забезпечення цілісності даних та носіїв інформації на ЕОМ.

У процесі супроводження робляться різноманітні зміни системи з метою:

- виправлення помилок і коригування програм, які видають неправильні результати;
- модернізації для додавання до системи нових можливостей або поліпшення якості вирішення окремих завдань відповідно до вимог замовника;
- адаптації системи до умов користувача, зумовленої змінами зовнішнього середовища й апаратури, де буде функціонувати система.

З цього випливає, що головна мета супроводження — доробка функцій та внесення змін у систему, якщо така потреба виникає в розробника.

Якщо до системи, що супроводжується, входить багато компонентів, то при коригуванні одного її компонента, як правило, зміни відносять до інших компонентів.

Для керування змінами створюється спеціальний контрольний орган — група, до якої користувачі подають пропозиції щодо змін, додавання або усунення деяких функцій у системі.

До завдань групи із супроводження системи входить також відслідковування стану кожного компонента, а також документації, що дає відповіді на такі запитання:

- коли і ким зроблено зміну;
- перелік змінюваних компонентів;
- правильність внесення зміни і хто засвідчив попередню зміну;
- хто заaproтоколював зміну;
- хто відповідає на запит про внесення змін і хто може скасувати його;
- чи пріоритетна зміна.

Усі ці питання належать до менеджменту, а не до технічних питань і мають регулюватися відповідною процедурою внесення змін. При цьому доцільно дотримуватися певних правил, а саме:

- кожній працюючій версії системи належить мати ідентифікацію;
- якщо версія модифікується, ідентифікатор ревізії присвоюється всім зміненим компонентам.

У такий спосіб групі супроводження протягом життєвого циклу завжди слід мати ідентифікатор версії й ідентифікатор ревізії кожного компонента та відповіді на наведені вище запитання.

При ухваленні рішення про внесення змін група проводить аналіз впливу змін на ризики, зокрема такі:

- неадекватність документації;
- неповнота внесених змін;
- погана структурованість коду;
- невідповідність стандартам;
- ускладнення проблеми;
- зростання незрозумілості коду;
- побічний ефект від внесення змін.

Усі ці аспекти збільшують вартість супроводження. У джерелах [1, 3] запропоновано діаграму процесів супроводження (рис. 10.5), в якій стрілки спрямовані до дій, котрі показують, що треба зробити, щоб вирішити, що саме змінити в супроводжуваній системі.

Корисно проводити горизонтальне трасування впливу змін у системі за стадіями життєвого циклу: вимоги → компонент проекту → реалізація проекту → тести.

Вертикальне трасування має відобразити взаємний вплив змін на одній стадії життєвого циклу, як-от: вплив нових вимог на попередні вимоги; для UML (див. главу 5) — вплив нових елементів моделювання на діаграми в межах одного виду.

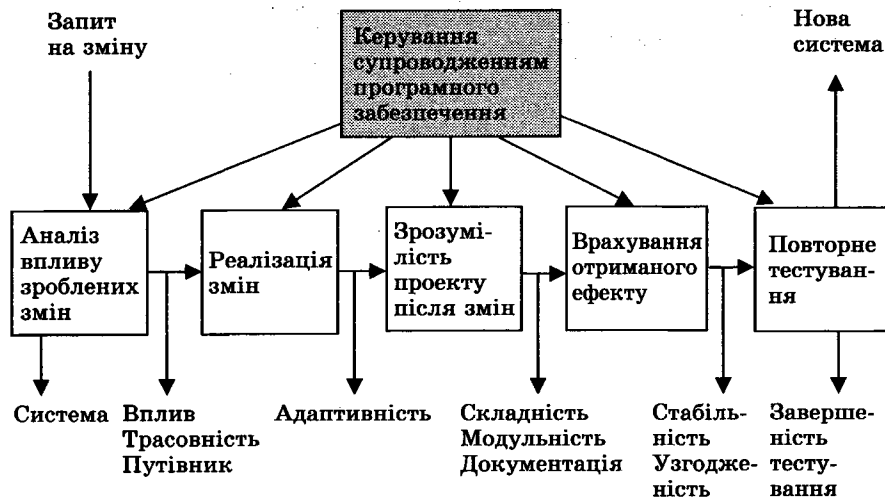


Рис. 10.5. Діаграма змін у процесі супроводження

Інструментальною підтримкою керування конфігурацією є: генератори крос-посилань, порівнювачі файлів і репозиторії керування конфігурацією.

### Контрольні запитання і завдання

1. Що розуміють під процесом керування конфігурацією програмного забезпечення?
2. Перелічіть основні завдання керування конфігурацією.
3. Дайте стислу характеристику понять ідентифікації, урахування статусу.
4. Які дії виконуються в процесі керування версіями програмного забезпечення?
5. Сформулюйте основні завдання керування супроводженням.

### Література до глави 10

1. Pfleeger S.L. Software Engineering. Theory and Practice. — Prentice Hall, 1998. — 576 p.
2. Buckle J.K. Software configuration management. — London: Macmillan Press, 1982. — 315 p.
3. Гласс Г. Руководство по надежному программированию. — М.: Финансы и статистика, 1982. — 256 с.
4. Лунаев В.В. Документирование и управление конфигурацией программных средств. Методы и стандарты. — М.: Синтег, 1998. — 215 с.
5. Лунаев В.В. Тестирование программ. — М.: Радио и связь, 1986. — 296 с.

## Глава 11. МЕНЕДЖМЕНТ ІНЖЕНЕРІЇ ПРОГРАМНИХ СИСТЕМ

### 11.1. Аналіз стану

Дослідження проблем планування й керування проектом показали, що в 1998 р. закінчилися невдало 26 % з усіх програмних проектів (порівняно з 40 % у 1997 р.), а 40 % значно перевищили кошторис або не функціонують (33 % в 1997 р.) [1]. Крім того, як відзначається, число невдалих проектів зменшилося в компаніях, котрі працюють над невеликими за об'ємом, а тому більш легкими для керування проектами.

Аналіз проектів, що зазнали краху, дозволив виділити 10 причин провалів [1], причому 7 із 10 цілковито залежали від рішень на початкових етапах перед тим, як було розроблено структуру або написано перший рядок програми. Нижче наводиться 10 причин невдач:

- 1) керівники проектів не розуміють вимог замовника;
- 2) масштаби проекту визначено неправильно;
- 3) зміни проекту провадяться з великими труднощами;
- 4) змінам піддають обрану технологію проектування;
- 5) замовник змінює вимоги;
- 6) взятий термін виконання проекту нереальний;
- 7) користувач не ухвалює деяких рішень;
- 8) інвестиції втрачено;
- 9) для реалізації проекту не вистачає виконавців;
- 10) менеджери проекту не застосовують прогресивних методів керівництва.

Для зменшення провалів проектів використовуються сучасні аналітичні методики й удосконалені мови розробки. Як показує досвід, розрекламовані технології вирішують завдання успішного створення якісного проекту в меншій мірі, ніж наведені чинники-рекомендації, а саме:

- а) проект починати з правильного кроку;
- б) підтримувати темпи роботи;
- в) забезпечити прогрес і правильні рішення;
- г) провести постаналіз (так званий "посмертний аналіз") завершених проектів.

Ці чинники зменшують провали програмних проєктів.

**Проект починати з правильного кроку.** Робота із створення якісного проєкту має завжди починатися за правилами, визначеними наведеними чинниками. Відомо, як важко виростити міцне дерево в неродючому ґрунті, аналогічно, майже неможливо привести до успішного завершення неправильного розпочату розробку.

Насамперед, до правильного кроку належить формування команди розробників шляхом добору кваліфікованих спеціалістів, у тому числі не більше 20 % “зірок”, здатних до ефективної сумісної роботи над проєктом (більша кількість “зірок” створює конфлікти). Крім “зірок”, до команди доцільно ввести надійних розробників із сумісними характерами і робочими навичками. Видатні члени команди — “зірки” — розв’язують складні питання, розробляють відповідальні алгоритми і проводять технічне навчання іншої частини команди.

У керівництва, розробників, проєктувальників і користувачів мають бути реальні завдання. Для уточнення всіх особливостей проєкту розробники сідають за стіл переговорів із замовником і складають об’єднаний документ. Бажано, щоб замовники і користувачі залучалися до роботи над проєктом і в майбутньому. Це не лише підвищує ступінь довіри між розробниками й користувачами, а й звужує сферу діяльності експертів-посередників.

Коли створено сильну команду, то наступний крок — створення умов для ефективної роботи і зменшення можливості виникнення конфліктів. Продуктивним середовищем для роботи команди є сучасні комп’ютерні засоби та зручності, ліцензійне програмне забезпечення, під яким розуміють не лише компілятори, а й системи аналізу та дизайну, реалізації, тестування, документації, графіки, а також особисті робочі місця й інші зручності (білі стіни, місце для зустрічей, як формальних, так і неформальних) тощо.

**Підтримка темпів.** Для збереження темпів роботи над проєктом необхідно сконцентруватися на трьох основних питаннях:

- зменшення плинності кадрів;
- контроль якості виконуваних робіт;
- керування процесом розробки, а не людьми.

**Плиність кадрів.** Це постійна проблема програмної індустрії, тому що переміщення або відхід окремого спеціаліста змушує інших членів групи вдаватися до трудомісткої неприємної роботи з вивчення незакінчених і не цілком задокументованих програм. Великий проміжок часу між відходом спеціаліста і знаходженням заміни може призвести до краху окремих частин проєкту.

**Контроль якості.** Цим необхідно займатися із самого початку розробки проєкту, встановивши процедури перевірки якості й знайшовши розробників, котрі можуть розробити високоякісний продукт.

**Керування процесом розробки.** Належить більше до процесу, ніж до людей. Створювана програма — це те, що продається. Тому у виконавців має бути свобода для приходу на роботу в нефіксований час і вільному одязі. Критиці піддаються результати праці, а не розробники.

**Підтримка прогресу і правильних рішень.** Програма відрізняється від інших конструкцій тим, що її не можна випробувати, доки її не закінчено повністю; її розробка починається з концептуальної моделі, а результати її застосування вимірюються після одержання продукту. Для підтримки й контролю прогресу, як правило, вибирається один із правильних рівнів деталізації дій під час проєктування з обов’язковим його виконанням.

Часто вибір технологій супроводжується хибними рішеннями без необхідного економічного й технологічного аналізу. Якщо ця технологія не відповідає вимогам ринку і її не підтримує солідна компанія, то проєкт створюється на піску.

**Постаналіз.** Для вивчення помилок робиться аналіз реалізації проєкту, щоб не повторити їх у новому проєкті. Аналіз дозволяє зрозуміти, чому неправильно складено кошторис, як команда і фірма розробляють програмні системи. Беручи до уваги, що кожна команда і фірма має свої особливості, які впливають на процес розробки, в процесі аналізу можна виділити певні закономірності особливостей ведення розробки та їхній вплив на отримані наслідки.

У джерелі [2] пропонується шість кроків проведення постаналізу проєкту:

- аналіз мети, поставленої на початку роботи;
- складання об’єктивного звіту всіма членами команди (і керівництва теж);
- організація збору даних, що сприяли керуванню, вимірюванню показників та критеріїв у процесі розробки і спілкуванню усередині груп і між групами;
- формування команди, яка виконає аналіз таким чином, щоб було представлено кожен групу розробників а керівництво — відсторонено;
- огляд проведеної роботи з аналізом допущених помилок за участю всіх зацікавлених осіб та його затвердження (всі учасники групи складають два списки: що відбулося і що не так, як передбачалося);
- повідомлення результатів аналізу всім учасникам команди розробників та керівництву;
- обговорення й затвердження рекомендацій для подальшої роботи.

Розглянуті чинники не повністю вирішують проблему керування проєктом, тому зупинимось на інших важливих його аспектах.

## 11.2. Планування

Складовою частиною роботи над проектом є план, в якому описується цикл розробки програмної системи із зазначенням фаз та станів проекту і представлення кожного з них у термінах виконуваних окремих завдань або процесів (діяльностей). При цьому відображаються зв'язки між процесами і визначається інтервал часу для виконання кожної діяльності, а також час її початку та завершення. До плану робіт включається створення різних видів демонстрацій для замовника: функцій, підсистем, правильності їх, надійності, безпеки, готовності. До документів, які виконуються за планом, належить комплект настанов користувача для виконання заданого набору обчислень, можливостей комунікації системи з іншими підсистемами тощо.

Важливим поняттям плану є так звані точки проекту на дузі часу виконання, які доцільно виділити як, наприклад, початкову і кінцеву точки функціонування процесу. Під час проведення аналізу проекту для упорядкування плану звичайно виділяються фази, кроки, діяльності (рис. 11.1).

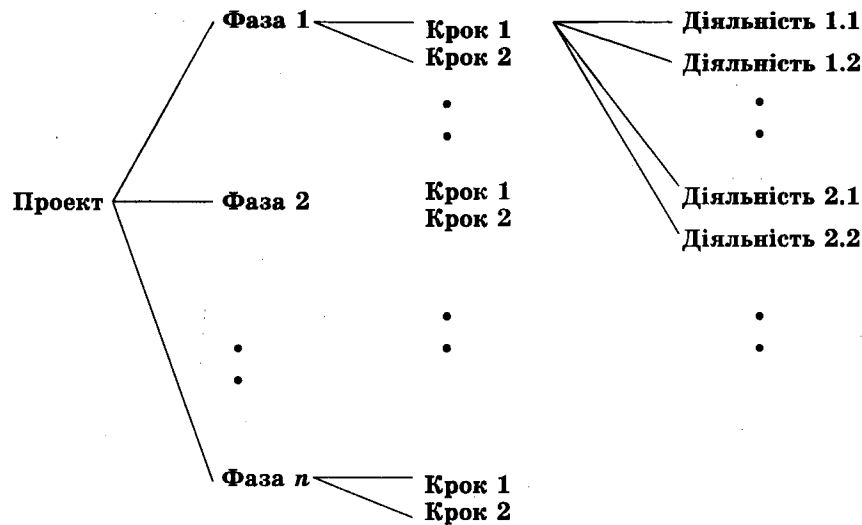


Рис. 11.1. Фази, кроки і діяльності в проекті

**Графи процесів проекту.** Робота із ведення проекту розподіляється на окремі фрагменти. У замовника має бути можливість контролювати проект у будь-якій його опорній точці, розглядати кожен окрему частину роботи, якщо частини можуть розвиватися паралельно.

Кожний процес описується за допомогою чотирьох основних параметрів:

- початкова позиція функціонування процесу (попередник);
- тривалість;
- термін;
- результат (кінцева точка).

*Попередник* — подія або набір подій, які відбулися до початку функціонування процесу і для яких описується набір умов початку процесу.

*Тривалість* — інтервал часу, за який процес має успішно завершити своє функціонування.

*Терміни* — дата, до якої процес цілком або частково завершує своє функціонування.

*Кінцева точка процесу* — контрольна точка, в якій замовник перевіряє якість результатів процесу, що завершив своє функціонування.

Граф процесів діяльності (рис. 11.2) складається з вершин, якими є основні точки проекту, поєднаних дугами, що відображають процеси. Такий граф дає інформацію про зв'язки між процесами проекту та порядок виконання їх за часом, включаючи можливі паралельні процеси ведення проекту. Відображення у графі паралельних робіт проекту — важливе і складне завдання, оскільки паралелізм на практиці може бути нездійсненним.

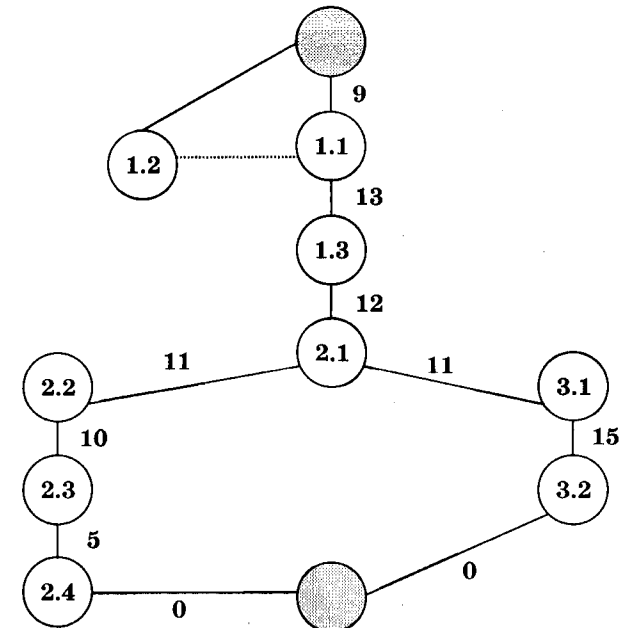


Рис. 11.2. Граф діяльності з часовими термінами на дугах

На дугах графу процесів можна додатково позначити інформацію про час, котрий передбачається для повного завершення функціонування даного процесу. Число на дузі вказує, наприклад, день завершення процесу. Дугам процесів, котрі входять до заключної вершини, відповідає часова позначка 0. Часова позначка дозволяє визначити на паралельних дугах, який з процесів виконується раніше (шляхом підсумовування числових часових позначок), а також порівняння заданого числа з тим, що відповідає паралельній дузі.

Аналіз графу проекту за даним методом називається методом критичної траєкторії (шляху). Він дозволяє оцінити кожний процес окремо і скласти порівняльні характеристики кількох конкурентних дуг проекту. Для проведення аналізу потрібно для кожного процесу і графу в цілому визначити за планом прогнозований і реальний час його виконання. Різниця між першим і другим приймається як “слабкий час”. Він обчислюється для всіх вершин графу у напрямку від початкової вершини по всіх дугах до кінцевої вершини і заноситься в таблицю для дуг.

У деяких планах присутні циклічні шляхи. У цьому разі аналіз критичних шляхів ускладнюється, тому що дані про тривалість кожного процесу завжди є приблизні. Іншими словами, за імовірною оцінкою визначається певний часовий інтервал, у межах якого процес функціонує.

**Кроки визначення плану.** Як правило, менеджер проекту описує проект у термінах п’яти кроків: планування, проектування, кодування, тестування та постачання. Далі розглянемо перші два кроки.

Система планування розподіляється на чотири види діяльності:

- визначення специфікацій;
- визначення бюджету;
- визначення розкладу;
- розвиток плану проекту в цілому.

Система проектування розподіляється на проектування на верхньому рівні, прототипування, проектування інтерфейсу користувача та представлення деталей проекту (див. главу 4).

Значна частина теперішніх засобів проектування націлена на візуалізацію структури проекту, процесів та кроків, причому паралельні діяльності та ступінь складності може бути виділено кольором або іконкою. Такі діаграми дозволяють менеджеру проекту побачити, які види діяльності мають виконуватися паралельно, і які кроки перебувають на критичному шляху.

Деякі діаграми проекту містять інформацію про ресурси. Наприклад, діаграма може демонструвати, як змінюється по місяцях число (за висотою стовпчика) працівників, котрі працюють над проектом, та інтенсивність роботи (колір стовпчика у відповідному місяці). На діаграмі також може бути відображено вартість задіяних у проекті ресурсів.

### 11.3. Персональна відповідальність у проекті

Найчастіше розподіл ролей співробітників у розробці проекту відповідає певним етапам розробки [5]. Нагадаємо їхній склад:

- аналіз вимог;
- проектування системи;
- кодування програми;
- тестування;
- підготовка і навчання;
- забезпечення якості.

Склад і кількість співробітників, котрі входять до групи проекту, залежить від масштабу робіт і досвіду співробітників. Важливо, щоб співробітники були настільки кваліфікованими, що могли б виявити помилки і неточності в проекті на найраніших стадіях ведення розробки. Поділ праці серед працівників за наведеними етапами має свої переваги, але потребує спеціальної техніки спілкування між групами працівників, щоб робота була ефективною (перевірки, перегляди, відкати назад, наскрізний контроль).

Найбільше підходять для виконання перелічених ролей працівники з такими ознаками:

- спроможність виконувати роботу;
- інтерес до роботи;
- досвід роботи з подібними програмними системами, інструментаріями, мовами й технологіями;
- досвід роботи в подібних середовищах функціонування;
- здатність до навчання;
- комунікативність з іншими співробітниками;
- спроможність нести відповідальність разом з іншими;
- професіоналізм у менеджменті.

Ці характеристики в деяких проектах можуть стати критичними, якщо не знати про спроможності того чи іншого працівника виконувати роботу чи бачити в цілому перспективи роботи з проектування або тестування системи. Працівникам однієї групи слід мати однакові погляди на ідеї та проведення дорученої їм роботи, а також на обрання стилю програмування. Відповідальність за розподіл великого фрагмента роботи на менші частини і призначення виконавців має бути чітко визначеною, а у відповідній документації визначено роль та відповідальність працівників.

**Стиль роботи.** Людям властиві різні стилі виконання роботи і стосунки з іншими працівниками [3]. Розглядаються працівники, які спочатку зважують усі деталі і збирають всю інформацію, а потім ухвалюють рішення з усіх питань, інші ж розбивають роботу на фрагменти й ухвалюють рішення окремо для прийнятого фрагмента. З іншого погляду, деякі працівники віддають перевагу формуванню ро-

бочого рішення шляхом викладення свого розуміння співробітниками та ухвалення рішення спільно з ними (робочий стиль *екстраверта*). Інші спочатку цікавляться думкою колег з того чи іншого питання, а потім самостійно ухвалюють рішення (*інтроверти*). Деякі працівники покладаються на свою інтуїцію і фаховий досвід для ухвалення рішення (*інтуїтивісти*), інші — керуються тільки раціональними і логічними доказами (*раціоналісти*). Природно, що в реальному робочому середовищі частіше трапляються змішані типи.

Раціональний екстраверт вважається гарним керівником. Він намагається обговорити проблему, але не дозволяє впливати на ухвалення остаточного рішення. Він рідко збирає багато інформації. Стиль його роботи — питати у своїх підлеглих лише про те, що стосується головної лінії ведення проекту (його не цікавлять подробиці, деталі документації тощо), але він не дозволяє впливати на ухвалення рішень та визначення термінів виконання.

Раціональний інтроверт уникає емоційних обговорень, йому необхідний час, щоб обдумати всі можливі шляхи розв'язання проблеми і прорахувати всі кроки. Він старанно зважає всі "за" та "проти", збирає всі факти. Репутація гарного працівника є для нього дуже важливою, він вважає, що робота має займати велику частину часу і вимагає цього від інших. Він акуратний і точний.

Інтуїтивний екстраверт часто ухвалює рішення на емоційному ґрунті. Намагається більше розповісти про себе і свої плани, а не вислухати інших. Часто базується на попередньому досвіді роботи, по натурі він випробувач. Йому важливо, щоб інші визнали його ідеї. Йому зручніше працювати в колективі, де встановлено і добре організовано зв'язки між співробітниками.

Інтуїтивний інтроверт — це творець. Але він починає творити тільки після того, як зібрав необхідну для себе інформацію. Уїнстон Черчілль належав до цього типу. Перед тим, як ухвалити рішення, він слухав і читав усі матеріали з певного питання. І часто ухвалював рішення, базуючись на своїх враженнях від почутого. Інтуїтивний інтроверт є гарним слухачем, намагається зібрати повну інформацію й ухвалити правильне рішення, звертаючи увагу не стільки на факти та об'єкти дослідження, скільки на зв'язки і відношення між ними.

У роботі над проектом необхідно враховувати стиль роботи не лише свого підлеглого, а й замовника. Якщо замовник інтроверт, то йому слід надавати більше інформації і часу на обмірковування для ухвалення рішення. З екстравертом потрібно більше спілкуватися, дозволяти висловлювати свої вимоги й ідеї. Інтуїтивісту слід підкидати більше нових ідей, заохочувати творчість і, якщо він раціональний, проводити для нього більше демонстрацій, базованих на фактах і схемах.

**Організація проекту.** Для належної організації ведення проекту підбирається прийнятна структура команди проекту на підставі таких даних:

- база і робочий стиль членів групи;
- число людей у групі;
- стилі роботи із замовниками і розробниками.

Один із популярних досі стилів ведення проекту уперше наводився при створенні IBM ОС—360 (рис. 11.3).

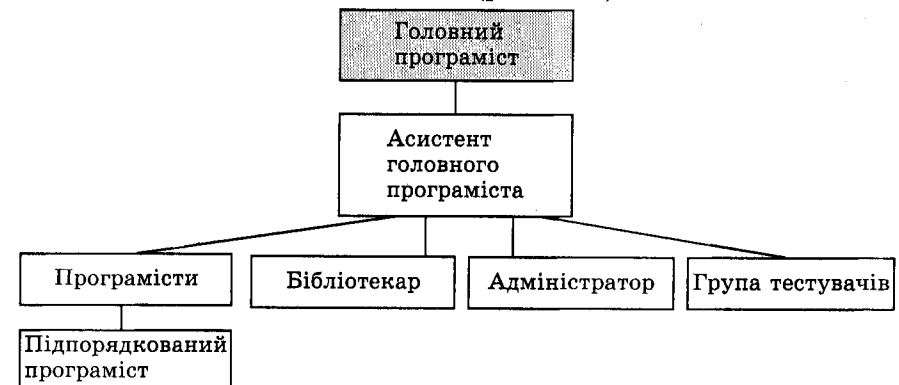


Рис. 11.3. Структура організації групи головного програміста

Головним, хто відповідає за проектування системи і ведення розробки, є керівник групи програмістів. Йому безпосередньо підпорядковуються програмісти, котрі мають право останнього слова в ухваленні рішень — головні програмісти. Головний програміст керує своєю підгрупою програмістів і безпосередньо посвячений у деталі проекту та розробки програми. Асистент головного програміста — той, хто дублює і виконує обов'язки головного програміста, коли це необхідно.

Бібліотекар — відповідальний за всю документацію проекту: за компіляцію та тестування всіх модулів, що тримаються в бібліотеці. Введення цієї посади дозволяє програмістам сконцентруватися на їхній безпосередній роботі, а не на пошуку помилок і необхідних матеріалів.

До групи входять адміністратор і група тестувачів, старші програмісти і молодші, які безпосередньо підпорядковані старшим. Хоча структура такої робочої групи ієрархічна, кожний член групи може спілкуватися безпосередньо з головним програмістом або з іншими співробітниками.

Головний програміст має самостійно переглядати частини проекту і програм. Іноді структура групи потребує переформування у разі зміни типу виконуваного завдання (наприклад, необхідно скоротити або збільшити адміністративну групу або групу, що тестує, і перекинути сили на іншу частину проекту). Такі рішення ухвалює головний програміст.

Альтернативну структуру ведення проекту описав Вейнберг у джерелі [1]. Це так зване знеособлене програмування, при якому в робочій групі всі несуть однакову відповідальність за якість продук-

ту. У проекті не концентруються на персоналіях, критиці піддається програмний продукт або інший результат, але не члени групи. Така структура підходить для маленьких груп програмістів з обмеженою відповідальністю. Але в деяких випадках підвищений рівень демократизації і децентралізованості може відіграти свою позитивну роль. Наприклад, коли при веденні розробки ще не відомі деякі вимоги, специфікації або нове апаратне забезпечення.

Подані вище два типи організації ведення проекту може бути скомбіновано один з одним.

#### 11.4. Проведення оцінок проекту

Однією з найважливіших робіт є оцінка кошторису проекту. Загальна ціна проекту складається з вартості його окремих частин: умови роботи, штат співробітників, використовувані засоби та інструменти. Ціна умов ведення проекту містить у собі сумарну ціну апаратного забезпечення, площі робочих кімнат, меблів, телефонів, модемів, канцелярських товарів і багато чого іншого, що підтримує фізичне середовище, в якому співробітники будуть працювати. Іноді таке середовище вже існує, іноді має бути створено додаткові умови (наприклад, безпеку).

Іноді не потрібно робити додаткові витрати на системи тестування, кодування або інші CASE-системи. Центральною оцінкою в проекті є оцінка зусиль з ведення проекту, що відображається в трудоднях штату працівників (так звані людино-дні) [4].

Всі коефіцієнти наведених нижче формул є емпіричними, тобто їх встановили експерти дослідним шляхом. Спеціалісти, котрі мають досвід в оцінці проекту, можуть оцінити його вартість до початку робіт з похибкою, меншою за 10 %.

Для проведення точніших оцінок програмним інженерам потрібно уточнити характеристики продуктивності для даного штату працівників, вимоги до проекту й інші чинники.

Правильність оцінки залежить від компетентності, досвіду, об'єктивності і сприйняття експерта. Метод побудови оцінки може бути або "зверху вниз", або "знизу вверх" від модулів до системи в цілому. При побудові нової системи вартість попередньої системи екстраполюється на нову з деяким коригуванням.

Коли такої можливості немає, то експертам пропонується зробити три оцінки: песимістичну ( $x$ ), оптимістичну ( $y$ ) і більш-менш реальну ( $z$ ). Тоді нормалізація цих індивідуальних оцінок може бути такою:  $(x + 4y + z)/6$ .

Delru-техніка [3] допускає різноманітні методи наближення узагальнених оцінок до реальних. Часто експерт спочатку проводить опитування всіх членів робочої групи і надалі виконує корекцію кожної оцінки, виводячи на їхній основі найбільш правдоподібну.

Експертний метод оцінки програмного продукту, наведений у джерелі [1], відображає досвід оцінки проектів однієї американської компанії програмістів. У цьому проекті рядок матриці містить ім'я модуля програмного продукту, а колонка відповідає ступеню складності його створення. Відповідно до матриці розрізняються такі завдання: нове ( $N$ ); старе ( $O$ ); легке ( $E$ ); поміrne ( $M$ ) і важке ( $H$ ). Аналізуючи матрицю, можна оптимізувати інтегральні оцінки.

В усіх наведених методах є свої недоліки. Часто важко визначити для кожного модуля, наскільки попередня система оцінок у цілому відрізняється від нової. Іноді система оцінок, яка успішно працює в одній компанії, не працює в іншій. У деяких співробітників групи песимістична й оптимістична оцінки можуть дуже відрізнятись.

**Алгоритмічні методи.** До них належать спроби побудувати модель, в якій відображаються зв'язки між витратами в проекті і чинниками, що на них впливають. Модель, звичайно, є рівнянням, в якому витрати — залежна змінна, а чинники, що впливають, — незалежні змінні. Наприклад, вартість проекту  $E = (a + bS^c) m(X)$ .

Тут  $S$  — оцінка розміру системи,  $a, b, c$  — емпіричні константи,  $X$  — вектор чинників вартості довжиною  $n$ ,  $m$  — регулюючий множник, оснований на цих чинниках. Чинниками можуть бути витрати проекту, які коригуються залежно від інших проектів, процесів, продуктів або характеристик ресурсів.

У джерелі [3] пропонується подібна модель як таке співвідношення:

$$E = 5.25 S^{0.91}.$$

Цю модель було використано для оцінки проектів побудованих систем довжиною від 4000 до 467 000 рядків коду, написаних 28 різноманітними мовами програмування високого рівня, на які витрачено 12 людино-місяців для 66 комп'ютерів.

У джерелі [5] Basley and Basili пропонують моделюючи техніку, названу позначкою-моделлю. Для побудови рівняння витрат враховуються характеристики організації-виконавця.

$$\text{Загальна формула } E = 5.5 + 0.73 S^{1.16}.$$

Далі з урахуванням можливої корекції через помилки будується початкова оцінка зусиль для створення продукту. Якщо  $R$  — коефіцієнт відношення реальних витрат  $E$  і прогнозованих витрат  $E'$ , тоді скориговане зусилля визначається такою формулою:

$$ER_{adj} = \begin{cases} R-1, & \text{якщо } R \geq 1 \\ 1-1/R, & \text{якщо } R < 1. \end{cases}$$

Далі вони коригують початкові витрати в такий вигляд:

$$E_{adj} = \begin{cases} (1+ER_{adj}) E, & \text{якщо } R \geq 1 \\ E/(1+ER_{adj}), & \text{якщо } R < 1. \end{cases}$$

У більшості моделей оцінка залежить від розміру системи (частіше за все дається в рядках коду). Модель СОСОМО Боема [2] увібрала в себе три техніки виміру проекту.

Боем розвив цю модель ще в 70-х роках, використовуючи показники ціни, а також інші оцінки — персонал, властивості проекту, продукту та середовища. Ця модель включає оцінку трьох стадій ведення проекту. На першій стадії звичайно будуються прототипи розв'язку задач підвищеного ризику (інтерфейс користувача, програмне забезпечення, система взаємодії, виконавські властивості тощо).

Оскільки на цій стадії розміри проекту передбачити важко, оцінки даються в термінах так званих об'єктних точок, як наприклад, число баз даних, число таблиць у базі даних клієнта, відсотки представлень екранів і повторне використання звітних форм, запозичених з попередніх проектів.

На другій стадії робиться оцінка витрат на проектування і реалізацію функціональних точок проекту, відображених у вимогах до проекту.

Третя стадія оцінки належить до завершення проектування, коли розмір системи може бути визначено термінами рядків програми й інших чинників.

Базовою моделлю служить таке рівняння:

$$E = bS^c m(X),$$

де первинна оцінка  $bS^c$  коригується вектором факторів, що впливають на вартість  $m(X)$ .

Далі ця модель розвивається в бік детального аналізу об'єктів, щоб врахувати скільки давніших об'єктів використовується для побудови нових і встановити число нових об'єктних точок, використовуваних для обчислення коефіцієнта продуктивності.

Параметр  $c$  у рівнянні змінюється від 0 до 1.0 для першої стадії і від 1.01 до 1.26 для останніх.

Для проведення оцінок проектів використовуються нейронні мережі як інструмент представлення видів діяльності для конструювання програмного продукту, комерційна модель Кемерера на основі регресивного дерева (більш докладно в Джерелі [3]) та інші.

## 11.5. Керування ризиком

Під час ведення проекту менеджер має передбачати можливість виникнення незапланованих подій, що можуть виникнути. Ризик — це небажана подія, яка може мати непередбачені негативні наслідки. Тому менеджерів слід уміти управляти ризиком.

Якщо в проекті ідентифіковано множину можливих подій ризику, які у свою чергу з великою ймовірністю можуть спричиняти негативні впливи, то такий проект називається *схильним до ризику*.

Наприклад, якщо вартість витрат, необхідних на мінімізацію негативних наслідків події, що відбулася, становить 50 000 гривень, а ймовірність ризику 0.3, то збиток ризику буде становити 15 000 гривень. Ймовірність ризику може змінюватися за часом, як і наслідки ризику. Тому до функцій менеджера проекту входять заходи з оцінки ризику та його наслідків у кількох часових точках ведення проекту, а також під час упорядкування плану дій у разі виникнення тієї або іншої небажаної події.

Є два основні типи ризику:

- загальний ризик, властивий усім типам проектів;
- ризик, специфічний для контролю конкретного проекту, що проявляється в кінцевих недоліках проекту (незавершеність проекту до обіцяного терміну тощо).

Є загальний ризик, який генерує ризик для всіх типів проекту. До такого ризику належить ризик, що виникає при неповному розумінні вимог, коли недостатньо професіоналів, за браком часу на тестування.

Ризик, специфічний для конкретного проекту, виражається в кінцевих недоліках проекту (незавершеність проекту до обіцяного терміну тощо).

**Діяльність керуванням ризиком.** Типами керування є ідентифікація ризику, його аналіз, визначення пріоритетів для кожного з виділених ризиків. Керування ризиком має важливі шляхи, наведені на рис. 11.4.

Перша частина — оцінка ризику проекту під час його розробки та підтримки. Оцінювання містить три діяльності: ідентифікація, аналіз ризику, визначення пріоритетів для кожного з виділених ризиків. Для ідентифікації ризиків використовуються різноманітні техніки.

Друга частина — контроль ризику. До нього належать зменшення ризику, планування та дозвіл (реакція) ризику.

Для побудови нової системи необхідно скласти перелік проблем і провести порівняльний аналіз усіх видів діяльності в проекті на кожному з етапів життєвого циклу. При розбитті процесу розробки на дрібніші фрагменти з'являється можливість виявити, на якому етапі виникає проблема, що спричиняє виникнення ризику.

Після ідентифікації ризиків будується схема пріоритету для ризиків відповідно до показників збитку ризику, оцінок наслідків ризику, оцінок ймовірності ризику.

Ризик знижується при поступовому нарощуванні функцій проекту і з проведенням регресивного тестування, за якого нова функція тестується разом з попередньою, що дає знижку ризику.

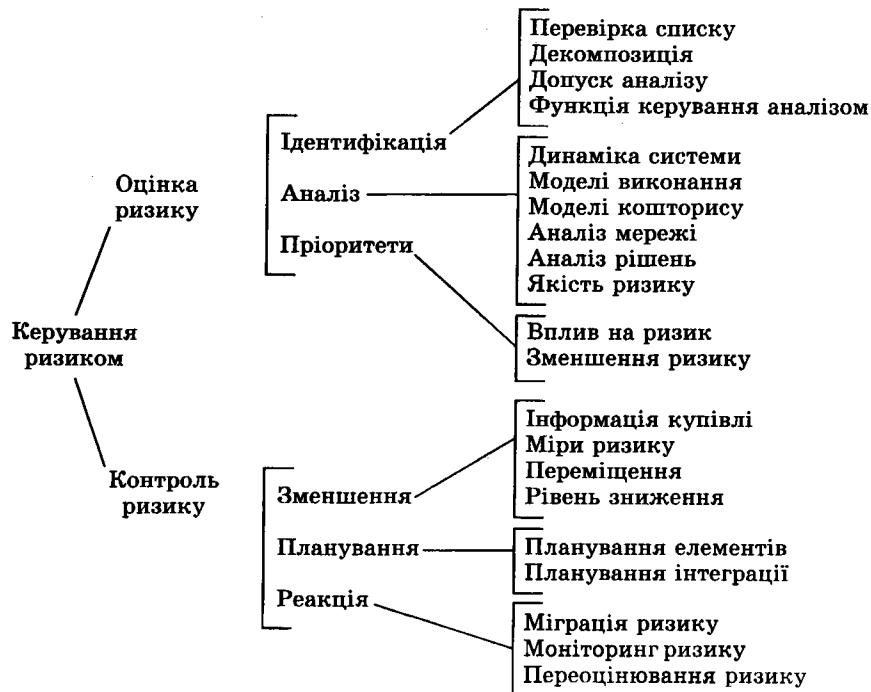


Рис. 11.4. Кроки керування ризиком

Для кожного можливого ризику визначається показник ступеня його ймовірності і показник втрат, пов'язаних з даним ризиком. Під час проведення регресивного тестування відшукуються критичні помилки. Залежно від того, наскільки ця помилка критична і від того, які показники ризику діють, обчислюється збиток ризику.

**Кроки керування ризиком.** Відповідно до діяльності керування ризиком (рис. 11.4) кроки керування можна розбити на такі етапи:

- зменшення ризику;
- планування ризику;
- вирішення проблеми ризику.

Для зменшення ризику є три стратегії, котрі визначають, як:

- уникати ризику шляхом зміни вимог до функціональності;
- перерозподіляти ризик шляхом його часткового розміщення в інших системах;
- відстежувати ризик й управляти наявними ресурсами.

Залежно від того, якими є масштаби можливого негативного впливу ризику на продукт і вартість витрат на його ліквідацію, визначимо систему керування ризиком як відношення

Збиток до мінімізації – збиток після мінімізації

Ціна проведення мінімізації ризику

Для мінімізації ризику є ряд стратегій. Наприклад, прототипування може допомогти обрати такий шлях розвитку проекту, на якому ризик зведений до мінімуму.

Боем у джерелі [2], наприклад, ідентифікував 10 точок ризику в проекті, які виникають найчастіше:

1. Скорочення штату або набір некваліфікованих працівників.
2. Нереалістичні проектні плани і бюджети.
3. Розробка функціонально хибних програмних елементів.
4. Розробка невдалого користувацького інтерфейсу.
5. Невдала постановка вимог.
6. Постійна зміна вимог.
7. Недоліки у внутрішній організації робіт.
8. Недоліки у взаємодії із замовником.
9. Невміння працювати в реальному часі
10. Обмежені комп'ютерні засоби.

## 11.6. План проекту

План проекту створюється для зв'язку із замовником, який може одержати необхідну інформацію з проекту або провести його інспекцію, і включає:

1. Визначення меж системи, ліній ведення проекту з точками (паралельно, послідовно) етапів розробки.
2. Формування часового плану (час виконання тих або інших робіт у проекті).
3. Структуру робочої групи і визначення періоду, типів робіт та ресурсів.
4. Технічний опис планованої системи з описом апаратного й програмного забезпечення проекту, компіляторів, інтерфейсів, системного устаткування, а також обмежень на час виконання та безпеку тощо.
5. Стандарти, процедури, запропоновані техніки та інструменти ведення проекту.
6. План досягнення якості.
7. План керування конфігурацією.
8. План підготовки документації з термінами виготовлення кожного документа та їхніх виконавців.
9. План керування даними і джерелами інформації із зазначенням відповідальних за формування дисків або дискет.
10. План тестування із зазначенням методів тестування, стадій і фаз.
11. План навчання із зазначенням документів, програмних систем, типів управління, використовуваних у процесі навчання.

12. План безпеки для визначення конфіденційності, обмежень на паролі і правила використання їх, у тому числі при інтеграції.

13. План керування ризиком.

14. План супроводження із зазначенням відповідальних за зміну кодів, ремонт устаткування, використання документації тощо.

Крім того, описуються алгоритми, інструменти, техніки перегляду або інспекції коду, мови ведення проекту або презентацій, мови кодування, техніки тестування.

## 11.7. Моделі процесу ведення проекту

У цьому підрозділі розглядаються такі аспекти ведення проекту, як вимоги до плану проекту, що можуть змінюватися залежно від того, наскільки вони піддаються впливу ризику, а також участь спеціалістів у проекті тощо [3]. Розглядається кілька успішних моделей ведення проектів (наприклад, фірми "Digital's Alpha AXP" і "F-16 aircraft software" [5]). Розглянемо ці аспекти на прикладах зазначених моделей.

**Керування реєстрацією.** На час створення великого проекту перша фірма витратила багато років, щоб побудувати так звану модель керування реєстрацією. Цей проект будувався на базі чотирьох операційних систем, у ньому брали участь 22 групи інженерів-програмістів.

Запропонована модель базується на таких положеннях:

- встановлення спеціального розуміння проблеми, що поділяється всіма учасниками проекту;
- створення повного і підтриманого всіма учасниками переліку зобов'язань співробітників;
- інспекції із системою зворотного зв'язку;
- заохочення кожного успіху в просуванні за проектом програм (метод Conklin, рис. 11.5) та в навчанні.

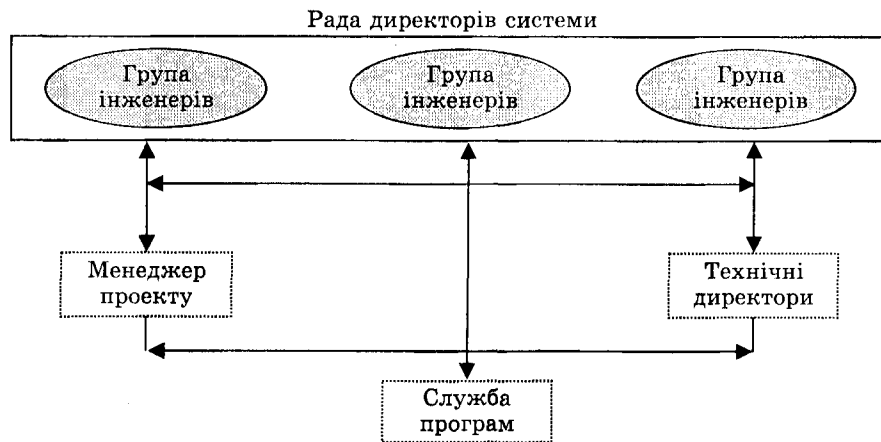


Рис. 11.5. Організація проекту

Передбачено, що проект ведуть кілька груп, реалізуючи одну загальну мету. Кожна група або підгрупа проекту визначає проект у термінах глобальних завдань. Менеджер розробляє плани і передає їх у групи. У планах зазначено зміст робіт для кожного завдання, відповідні коментарі й обмеження на виконання плану відповідно до списку рішень. Під час аналізу результатів робіт дається оцінка внеску кожного в спільну роботу. Деякі члени команди можуть не брати участі в проведенні конкретних робіт, але бути відповідальними за роботу.

Крім того, менеджер постійно інспектує роботи проекту, поставлення їх і чекає від виконавців повідомлень про "критичні події", що можуть спричинити ризик. У цьому разі він проводить необхідні зміни для просування проекту далі, фіксує кроки просування і відзначає персоналії. Менеджер з'ясовує наявні труднощі, засоби подолання їх і плани подальшого поліпшення проекту.

До переліку завдань менеджера входить координація груп апаратного й програмного забезпечення з визначенням обов'язків окремих співробітників у технічному проекті. Оскільки координувати роботу у великому проекті дуже важко, то менеджер проекту і технічні директори створюють кілька планів, добре скоординованих один з одним. Співробітники можуть працювати і з загальним планом, і з більш докладними інструкціями.

Для запобігання непередбаченим подіям ризику менеджер інспектує проведення робіт, унаслідок чого робиться короткий звіт, що встановлює:

- додержання термінів виконання плану;
- віхи виконання плану;
- шляхи в проекті, піддані критиці;
- планування необхідної діяльності на критичному шляху;
- питання, які виникли і які розв'язано;
- питання, які виникли, але які не розв'язано.

**Моделювання.** У джерелі [3, проект F-16] подано модель відповідальності членів команди інтегральної розробки модульної програмної системи з нарощуванням можливостей і розширенням функцій проекту, а також з визначенням вартості і планів змін у проекті. Модель відповідальності команди наведено на рис. 11.6.

Менеджер керує розподілом обов'язків і встановлює терміни для виконання реалізації трьох однакових за обсягом завдань.

Учасники проекту працювали в матричній організації, в такій, де кожен інженер входив до складу визначеного типу робіт (проекування або тестування) в одному або більше проектах. Суть організації робочої групи інтегрованого розвитку продукту полягає в можливості працювати спільно, відповідно до загальних законів дисципліни для всіх груп та окремих засобів контролю за роботою.



Рис. 11.6. Модель відповідальності команди в інтегрованому проекті

У наведеній моделі група — це комбінація працівників, відповідальних за результат своєї роботи. Виразник інтересів — це той, хто замовляє проект і впливає на результат або на вибір шляху для досягнення результату. Під час розробки обов'язки і ролі працівників постійно змінюються, що зручно для проекту, в якому необхідне проведення оперативних процедур, і часто змінюються плани. Для планів встановлюються терміни в межах тижня або навіть годин (а не місяця, як це було розглянуто в попередній методиці організації робіт).

З метою організації робіт великого колективу використовуються карти обов'язків, схеми, що відбивають терміни на шляху виконання робіт для кожної частини проекту. Показником того, наскільки виконано завдання, є діаграма планованих і реально виконаних робіт. Часто ця модель обов'язків об'єднується з ручним моделюванням для уникнення небажаних сценаріїв та зразків взаємодії між працівниками.

**Встановлення віх.** Віхи на шляху виконання встановлюються за такими основними положеннями процесу розробки програмного забезпечення:

- завдання життєвого циклу;
- архітектура життєвого циклу;
- початкові операційні можливості.

Основні завдання життєвого циклу пов'язані з метою виразників інтересів замовника. Замовник визначає межі системи, середовище і зовнішні системи, з якими система має взаємодіяти. Потім замовник за допомогою сценаріїв визначає, як система буде використовуватися. Сценарії потрібно виражати, вдаючись до термінів прототипів, екранів, потоків даних тощо, а також супроводжувати прикладами по-

ведінки системи в критичних випадках. Для побудови початкового плану життєвого циклу за джерелом [2] треба розв'язати такі основні питання.

1. Для чого система розробляється.
2. Що і коли буде зроблено.
3. Хто відповідальний за виконання функцій.
4. Як буде виконуватися робота технічно і яка стратегія керування перебігом робіт.
5. Які ресурси необхідні.
6. Чи є сенс займатися цим.

Для ведення проекту здійснюється підхід, оснований на спіральній моделі життєвого циклу. Зміст цієї моделі полягає в тому, що на кожному витку процесу розробки робляться припущення про завдання, умови вибору їх та обмеження, з якими зіткнуться розробники на новому витку спіралі.

## 11.8. Метрики менеджменту

Менеджмент програмної інженерії має спиратися на певні кількісні показники процесу, які будемо називати метриками. Слід зазначити, що в програмній інженерії немає визнаної сталої системи метрик, однак є багато пропозицій, аналіз яких може слугувати стартовою вхідною позицією для системи вимірів, адекватної конкретному спектру розробок.

Будемо розрізняти метрики процесу розробки і продукту розробки.

У ролі метрик процесу може бути використано міру трудовитрат [4] (наприклад, людино-місяці), час розробки, кількість помилок, знайдених під час тестування. Зокрема, щодо сценарного підходу, викладеного в п. 3.5, може бути названо такі метрики процесу:

- загальний час розробки;
- термін розробки для кожної стадії;
- термін модифікації моделей попередньої стадії;
- термін виконання кожної роботи:
- 1) специфікація сценаріїв;
- 2) специфікація об'єктів;
- 3) конструювання сценаріїв;
- 4) конструювання об'єктів;
- 5) тестування сценаріїв/об'єктів;
- кількість знайдених помилок під час ревізії;
- кількість змін у моделі попередньої стадії;
- вартість забезпечення якості;
- вартість введення нового процесу розробки.

Ці міри є базисом для планування майбутніх розробок. Показники міри залежать від проекту, організації, виконавців, складності завдання.

Метрики продукту може бути запропоновано для кожної стадії розробки. Загальною мірою є трасованість. Вона може визначатися як число первинних вимог прямо успадкованих у моделі сценаріїв (див. главу 5).

Прикладами метрик стадії аналізу є локальність вимог для сценаріїв, а також кількість вимог; акторів та сценаріїв; об'єктів кожного типу; об'єктів, які діють у сценарії (максимальне, середнє); операцій об'єкту; стимулів в одному сценарії; параметрів стимулу.

Спеціальною мірою (рівень повторного використання) є відношення розміру робочого продукту, створеного з готових компонентів, до розміру прикладної системи в цілому. Ця міра може враховуватися для визначення вартості та якості розробленого програмного забезпечення.

До метрик стадії конструювання належать загальне число класів, загальне число операцій, а також число використаних готових та нових класів; повторно використовуваних операцій; стимулів, що відправлено; класів, які успадковують (або використовують) специфічну операцію; класів, від яких залежить той чи інший клас; класів, які залежать від окремого класу; прямих користувачів класу чи операції (кандидата на роль компоненти), а також ширина і глибина ієрархії успадкування.

Крім того, є статистичні метрики, які використовуються для оцінки деяких характеристик об'єктів. До статистичних метрик належать середня кількість операцій у класі; середня довжина операцій (в операторах); середня кількість стимулів, успадкованих операцій; середня кількість спадкоємців класу; середня кількість спадкоємців операції.

Усі перелічені міри є трохи суб'єктивними. Вони спираються на суб'єктивні оцінки експертів, тому кожна організація, кожний конкретний процес мають зафіксувати використовувані ними міри, які дозволять оцінити доцільність менеджменту програмних систем.

## Контрольні запитання і завдання

1. Які завдання вирішуються менеджментом інженерії програмних систем?
2. Дайте визначення процесу планування менеджменту програмного забезпечення.
3. Які підходи до організації розробки Вам відомі?
4. Які є методи визначення вартості програмного забезпечення?
5. У чому суть керування ризиком при створенні програмних систем?
6. Наведіть приклади стратегій керування проектом.
7. Які Ви знаєте метрики процесу керування проектом?

## Література до глави 11

1. *Reiter D.J.* Software management, IEEE Computer Society Press, Los Alamos, 1993.
2. *Бозм Б.У.* Инженерное проектирование программного обеспечения. — М: Радио и связь, 1985. — 511 с.
3. *Glib T.* Principles of software engineering management. — Wokingham, England: Addison-Wesley, 1998. — 396 p.
4. *Андон Ф.И., Суслов В.Ю., Коротун Т.М., Коваль Г.И., Слабосницка О.О.* Визначення витрат на створення програмного забезпечення автоматизованих систем // Проблемы программирования. — 1998. — Вып. 3. — С. 23—34.
5. *Pfleeger S.L.* Software Engineering. Theory and Practice. — Prentice Hall, 1998. — 576 p.

## Глава 12. ПОВТОРНЕ ВИКОРИСТАННЯ В ПРОГРАМНІЙ ІНЖЕНЕРІЇ

### 12.1. Зміст проблеми

Однією з характерних рис інженерної діяльності є використання готових рішень або деталей. Однак усі, хто працює над створенням реальних систем, знають, що промислове використання готових рішень у програмній інженерії ще не стало повсякденною практикою. Якщо у світі працюють майже 8 млн програмістів, то приблизно 80 % з них працюють над створенням програм обліку й організаційного управління на кількох рівнях: окремого підрозділу фірми, окремого аспекту діяльності фірми, фірми в цілому, корпорації, галузі і, нарешті, держави. Це, переважно, задачі розрахунків, статистики, допомоги в прийнятті рішень при управлінні різноманітними ресурсами — кадровими, фінансовими тощо.

За оцінками експертів, 75 % цих робіт у світі дублюються — на тисячах підприємств створюються програми складського обліку, нарахування зарплати, розрахунку витрат на виробництво продукції, складання маршрутів деталей на виробничому конвеєрі тощо. Хоча більшість із цих програм типові, але кожного разу знаходяться особливості, що не дозволяють застосувати розроблену раніше програму. Тому нині активно розвивається напрям, водночас науковий та інженерний, який названо повторним використанням або компонентною розробкою програм.

*Компонентна розробка* — це метод побудови програмного забезпечення з конструкцій за каталогом — як композиції готових компонент. Йдеться не лише про порції програмного коду або програмні модулі.

*Повторне використання (ревикористання)* — це використання для нових розробок будь-яких порцій формалізованих знань, здобутих під час реалізації завершених розробок програмних систем.

Накопичений досвід розробки систем програмного забезпечення може бути зафіксовано в різноманітних формах, починаючи від конкретних параметризованих програмних модулів і кінчаючи програмними архітектурами та середовищами. Далі будемо називати *повтор-*

*но використовуваними компонентами (ПВК)* елементи знань про минулий досвід розробки систем програмування, якщо:

- 1) їх можуть використовувати не лише їхні розробники;
- 2) їх можна адаптувати для створення нових систем.

Зауважимо, що при цьому потрібен каталог, з якого можна зрозуміти, які деталі є і як їх можна поєднувати в конструкцію. Тож повторне використання має включати систематичну цілеспрямовану діяльність зі створення каталогу [1]. Ми називаємо таке повторне використання *систематичним*, підкреслюючи цим, що йдеться не про компоненти, створені внаслідок розробки програмної системи, після завершення якої “випадково” з’ясувалося, що ці компоненти можуть ще комусь стати в пригоді. Досвід так званої реінженерії готових програмних продуктів показав безперспективність пошуку таких знахідок. На відміну від нього, *систематичне повторне використання* є капіталомістким підходом, що передбачає наявність двох явно виділених процесів в життєвому циклі розробки програмних систем. Зупинимось на суті цих процесів.

*Перший процес — створення ПВК.* Він включає такі кроки:

- 1) вивчення спектра завдань, що вирішуються, виявлення серед них загальних підходів до вирішення;
- 2) побудову для них компонент, які реалізують знайдені підходи або окремі їхні елементи, котрі ми назвали повторно використовуваними компонентами;
- 3) побудову каталогу, націленого на пошук необхідних компонент.

Для успішної реалізації цього процесу необхідно мати певний досвід у вирішенні не одного, а кількох подібних завдань, що дозволяє виявити як їхні спільні риси, так і розбіжності, щоб знайти загальне вирішення для їхньої реалізації, а також способи налаштування на характерні для кожного завдання особливості.

*Другий процес — конструювання цільових систем з готових компонент.* Він передбачає такі кроки:

- 1) зрозуміти, що має робити нова цільова система, для чого вона створюється і які вимоги до неї ставляться;
- 2) знайти у каталозі серед готових компонент ті, які вважаються підходящими, і зрозуміти, що вони роблять;
- 3) зіставити мету нової розробки з можливостями знайдених ПВК і прийняти рішення про доцільність використання їх;
- 4) застосувати відібрані ПВК й інтегрувати їх до нової розробки, забезпечивши необхідні сполучення.

Створення ПВК потребує вкладення капіталу, а використання дозволяє отримати зиск за рахунок економії трудозатрат. Як і всі інвестиції, інвестиції в повторне використання потребують дослідження й оцінки ефективності вкладень капіталу, прогнозування термінів та обсягів повернення цих вкладень, оцінки ризиків та інших традиційних

бізнес-процесів. Інакше кажучи, бізнес повторного використання, як будь-який бізнес, потребує спеціальних зусиль для менеджменту всіх його процесів. Критеріями успіху такого бізнесу є:

1) забезпечення повторного використання меншими трудовозатратами, ніж розробка програмних систем як разових продуктів;

2) забезпечення пошуку придатних для використання компонент меншими трудовозатратами, ніж нова розробка їхніх функцій для потреб системи, що проектується;

3) забезпечення настройки компонент на нові умови меншими трудовозатратами, ніж нова розробка.

Провідні світові виробники програмного забезпечення широко інвестують дослідницькі проекти з технологій повторного використання та накопичення ПВК. Програми розвитку повторного використання досліджуються і на загальнонаціональному рівні (відомі загальнонаціональні програми Японії, Голландії, Великобританії, Італії; американські програми департаменту оборони, NASA, військово-повітряних сил США тощо), і на рівні провідних світових корпорацій ("IBM", "ERICSSON", "MICROSOFT", "HEWLETT-PACKARD" та інші).

У подальших підрозділах цієї глави досліджуються механізми створення ПВК, класифікації їх, пошуку й використання.

## 12.2. Визначальні властивості ПВК та їх типові сполучення

В основі створення ПВК лежить ідея виявлення так званих родових знань про проблеми, які розв'язуються програмними системами. Цим терміном позначаються знання, котрі сприяють вирішенню кожного з певного кола завдань (таке коло визначає певну проблемну галузь). Родові знання виявляються як результат абстракції артефактів програмного забезпечення. Під *артефактами* мають на увазі будь-які продукти діяльності фахівців з розробки програмного забезпечення (ПО).

Типи артефактів при цьому не обмежуються фрагментами коду. Вони можуть бути структурами проекту, структурами реалізації модулів, документації, трансформації і т.д.

За величезної різноманітності форм ПВК, які використовуються у комп'ютерній практиці, є безперечна спільність у техніці їхнього застосування. Всі вони конструюються в процесі абстрагування і застосовуються шляхом виконання процесів селекції, спеціалізації й інтеграції.

Перелічені чотири процеси визначають, так би мовити, чотири виміри, в аспектах яких ПВК можуть абстрагуватися, класифікуватися, добиратися і зіставлятися. Розглянемо ці процеси докладніше.

**Абстракція.** Абстракція є визначальною властивістю ПВК, кожна абстракція — продукт аналізу ознак зібрання сутностей, для якого виділяються суттєві спільні ознаки й відкидаються несуттєві з

певного погляду. Внаслідок цього визначається абстрактна або родова сутність, що матеріалізується як ПВК.

Кожна абстракція може розглядатися як така, що має дві частини — видиму та приховану. Видима частина є специфікацією тих знань, які необхідні для використання ПВК. Прихована частина містить деталі реалізації ПВК, невидимі на рівні її специфікації.

Видиму частину, у свою чергу, доцільно розглядати як сукупність стабільної або фіксованої та змінної складових.

Змінна частина представляє змінні властивості ПВК, її здатність адаптуватися до якогось спектра застосувань, фіксована частина — інваріанти. Тобто, специфікація абстракції із змінною частиною відображається в множину альтернативних реалізацій.

Розподіл абстракції на видиму фіксовану і змінну та приховану частини є не внутрішньою властивістю абстракції, це передовсім проектні рішення розробника абстракції, які він приймає, даючи відповіді на такі запитання:

— яка інформація потрібна користувачеві? Потрібна користувачеві інформація подається як специфікація (видима частина);

— які з властивостей абстракції користувачеві доведеться змінювати? Такі властивості доцільно подати у змінній частині специфікації.

Як приклад, розглянемо стек — відому схему запам'ятовування елементів за принципом: "елемент, що надійшов до сховища даних останнім, видається першим". Абстрагуємося при цьому від структури та форми подання елемента стеку. Опис принципу роботи стеку подається в специфікації, а максимальна глибина стеку може бути або в прихованій частині (коли користувач нею не цікавиться), або в змінній частині (коли необхідно міняти її за бажанням користувача). Якщо подати глибину стеку у фіксованій частині, користувач буде знати її значення, але не зможе його змінити.

**Добір або селекція.** Ключовою особливістю технологій використання ПВК є та обставина, що, приступаючи до нової програмної розробки, користувач не має відомостей про готові компоненти, тому у ПВК має бути чітка і виразна специфікація абстракції, котра давала б користувачеві можливість ефективно переглядати, розуміти, зіставляти і добирати підходящі для нього ПВК із числа наявних.

Подібно до бібліотечного пошуку, на допомогу користувачеві може бути надано спеціальні каталоги й класифікатори зібраних компонент, призначених здійснювати навігацію пошуку і полегшувати розуміння сутності ПВК.

**Спеціалізація.** Як було зазначено вище, зазвичай ПВК є матеріалізацією певної абстракції або родового артефакту, який неможливо безпосередньо використати. Для його використання необхідно спеціалізувати або конкретизувати його, тобто провести операцію, обернену до абстракції, зміст котрої полягає в тому, щоб до родових ознак,

властивих кожному з “представників роду”, додати ознаки конкретного застосування — параметри, трансформації, обмеження тощо. Родовий артефакт є, по суті, абстракцією зі змінною частиною. Спеціалізація родового артефакту — це отримання реалізації абстракції на основі довідзначення змінної частини ознаками конкретного застосування.

**Інтеграція.** Для ефективної інтеграції розрізнених артефактів, втілених у ПВК, до конкретної програмної системи потрібно, щоб користувач розумів інтерфейс артефакту, тобто особливості його взаємодії з іншими артефактами або з якимсь каркасом, який до певної міри можна вважати базою інтеграції. Можна сказати, що інтерфейс артефакту — це абстракція, в якій не враховуються внутрішні властивості артефакту, специфікація котрої визначає правила взаємодії артефактів між собою.

Для оцінки ефективності абстракції вводиться інтуїтивна міра, яку названо *когнітивною відстанню*. Цим терміном позначається кількість інтелектуальних зусиль, які необхідно витратити розробникові програмної системи для того, щоб перевести її з однієї стадії життєвого циклу в іншу. Для створення програмної системи за допомогою ПВК розробник ставить за мету мінімізацію когнітивної відстані між початковою концепцією системи та її кінцевою реалізацією, готовою до виконання.

Засобами досягнення цієї мети є:

- використання фіксованих і змінних абстракцій, лаконічних і виразних водночас;
- максимізація прихованої частини абстракції;
- застосування автоматичного відображення специфікації абстракції в її реалізацію (наприклад, шляхом компіляції).

Якщо поглянути на розвиток у часі засобів створення програм з погляду повторного використання, то історична картина виглядає як поширення застосувань комп'ютера на нові і нові домени, набуття практики вирішення окремих завдань в рамках цих доменів, узагальнення набутого досвіду як накопичення типових абстракцій та втілення їх у ПВК певної форми. Історично першим доменом застосування програмних засобів була галузь числових розрахунків, для якої було створено і перші ПВК, котрі мали вигляд спочатку типових програмних модулів (так званих підпрограм — аналог сучасних бібліотечних функцій), а згодом і мов програмування. Перше покоління мов програмування було втіленням типових абстракцій алгоритмічних обчислень (мови програмування Фортран, Алгол-60 та подібні їм).

Друге покоління з'явилося як відповідь на поширення комп'ютерних методів на домени обробки великих обсягів економічної інформації, завдяки чому було виділено абстракції типових процесів обробки даних, збирання програм з окремих модулів (Кобол, PL/1, Паскаль, Модуль тощо).

Поява візуальних можливостей обміну інформації з комп'ютером покликала до життя наступні покоління ПВК, представлені абстракціями комп'ютерної графіки, діалогової взаємодії, узагальненою концепцією інтерфейсу. Ці узагальнення втілено в мови програмування (C++, Java тощо) та багато інших форм представлення узагальненого досвіду розв'язання певних проблем у вигляді ПВК. Нижче досліджуються різноманітні форми втілення ПВК у сучасній комп'ютерній практиці.

### 12.3. Поширені в користуванні категорії ПВК

У сучасній технології програмної інженерії використовуються категорії ПВК, кожна з яких є типовим сполученням наведених вище аспектів розгляду ПВК, а саме:

- типу абстракції, шляхом якої виділяються родові знання, готові до реторикостання в кожній задачі з певного спектра;
- механізмів спеціалізації або настройки родових знань на конкретні вимоги стосовно створюваних або цільових програмних систем;
- механізмів інтеграції ПВК у цільові системи;
- механізмів вибору серед готових конкуруючих ПВК, тих, що відповідають вимогам цільової розробки і забезпечують мінімізацію когнітивної відстані для певних фаз життєвого циклу розробки. Експертне дослідження сучасного стану повторного використання дозволило виділити типові категорії ПВК, які значно поширені в сучасній практиці програмних систем. Їхні особливості обговорюються в п. 12.3.1—12.3.8.

#### 12.3.1. Мови програмування високого рівня

Конструкції мов програмування є **абстракціями** типів даних, функцій обробки, способів конструювання алгоритмів та збирання програм з окремих конструкцій, характерними для окремих проблемних галузей або більш-менш широкого спектра проблемних галузей. Таким чином, конструкції мов програмування відіграють роль ПВК.

Конструкції мов програмування високого рівня — це специфікації, котрі мають реалізацію в асемблері.

Постійна частина абстракції відповідає семантичному опису конструкцій мови на рівні метасимволів, змінна частина представлена термінальними символами конструкцій.

Прихована частина абстракції містить подробиці реалізації конструкції в асемблері, наприклад, проміжні дані тощо.

Відображення специфікації в реалізацію виконується цілком автоматично за допомогою компілятора.

**Спеціалізація** конструкцій здійснюється шляхом підстановки термінальних виразів (наприклад, арифметичних або умовних) в граматичні конструкції мови.

**Селекція** підходящої мови програмування серед мов-претендентів здійснюється шляхом експертного зіставлення конструкцій мови-претендента з інтуїтивним уявленням щодо функціональних та виконавчих властивостей тих програмних систем, які потрібно розробити.

**Інтеграція** окремих конструкцій кількох мов у нову мову програмування здійснюється як результат неформалізованої інтелектуальної діяльності автора-експерта. Прикладом може бути інтеграція конструкцій мов Алгол-60, Фортран, Кобол у мови PL/1, Алгол-68, Паскаль та ін.

### 12.3.2. Компоненти вихідного коду

Цим терміном назвемо фрагменти коду вихідною мовою програмування, які створювалися і запам'ятовувалися спеціально для повторного використання.

До цієї категорії належить абсолютна більшість ПВК, котрі на сьогодні масово використовуються. Бібліотеки функцій та класів для багатьох мов програмування налічують тисячі компонент.

Природа ПВК значною мірою залежить від вихідної мови. У багатьох із сучасних мов програмування передбачено спеціальні засоби для подання абстрактних типів даних, абстракцій управління, процедур, модулів, пакетів, функцій, класів тощо.

Головна проблема у створенні бібліотеки ПВК — знайти точні специфікації **абстракції** для компонент. Без цього користувачеві для вибору ПВК буде важко зрозуміти призначення компоненти, а іноді для такого розуміння навіть доведеться вдаватися до аналізу власне коду компоненти, що зведе нанівець переваги повторного використання.

Розробник бібліотеки ПВК має забезпечити специфікацію абстракції, яка стисло подає поведінку компоненти, схему її класифікації та пошуку. Найбільшого успіху досягли бібліотеки з абстракцією “одним словом”, що є однозначно зрозумілим у межах проблемної галузі, наприклад, *матриця* та *SIN* у числових обчисленнях, абстракції схем запам'ятовування Буча *стек*, *черга*, *дерево* в системному програмуванні. Використання компонент вихідного коду часто вимагає роботи з усіма частинами абстракції — фіксованою і змінною, прихованою і видимою.

**Спеціалізація** компонент може відбуватися кількома способами. Найдавніший з них і досі має застосування — це пряме редагування вихідного коду, для чого доводиться доходити до подробиць низького рівня. У цьому разі ефективність застосування ПВК досить мала.

Більш ефективний підхід — проектування родових компонент з передбачуваними правилами адаптації до вимог конкретного застосування, яке може виконуватися автоматично. Тоді замість прямого редагування коду розробникові достатньо визначити родові параметри як приховану частину абстракції специфікації.

Наприклад, для сортування певних сутностей параметрами можуть бути типи елементів, що підлягають сортуванню, їхня часткова упорядкованість, схема індексування, яку було застосовано під час накопичення їх. Процес спеціалізації при цьому може виконуватися автоматично, як, наприклад, під час використання макрозасобів з параметрами і відомих родових пакетів АДА.

**Селекція** ПВК вимагає спеціальних способів і зусиль як з боку організаторів бібліотек вихідних модулів, так і з боку користувачів. Зазвичай застосовуються поширені в бібліотечній справі способи, як, наприклад, ієрархічна рубрикація, використання індексів ключових слів, котрі характеризують зміст бібліотечної компоненти, неформальні анотації. Для користувача пошук компоненти при цьому зводиться до пошуку рубрики, що відповідає характеру завдання, яке він має вирішувати (наприклад, обчислювальні функції, робота з потоками даних, виведення на екран тощо), пошуку серед визначених тем рубрики і вивчення неформальних анотацій.

Для **інтеграції** ПВК більшість компіляторів з мов програмування мають автоматичні засоби збирання модулів та вирішення можливої колізії імен.

### 12.3.3. Класи об'єктів та абстрактні класи

Згідно з парадигмою об'єктного підходу, наведеною в п. 3.3.1, програмна система розглядається як спільнота взаємодіючих об'єктів, істотними ознаками яких є:

- стан, представлений сукупністю атрибутів;
- поведінка;
- спроможність вступати у відношення з іншими об'єктами;
- спроможність посилати одне одному повідомлення.

Об'єкти з однаковими істотними ознаками об'єднуються в класи. Клас, таким чином, — це **абстракція** властивостей, поведінки та інтерфейсів об'єктів. Інтерфейс є видимою частиною абстракції, що визначає правила взаємодії об'єкта із зовнішнім світом, тобто фактично правила його використання.

Об'єкти можуть визначатися на стадії аналізу вимог до розробки системи, тоді вони відповідають реальним об'єктам предметної області або її поняттям. Повторне використання таких об'єктів на ранніх стадіях життєвого циклу розробки програм у цьому разі не залежить від середовища реалізації системи. У процесі реалізації властивості об'єкта, як правило, відображаються в атрибутах відповідних баз даних, а поведінка об'єкта відображається у визначені для нього операції або методи. Інтерфейси між об'єктами представляються тими повідомленнями, які вони можуть посилати.

Для об'єктно-орієнтованих мов програмування розроблено представницькі бібліотеки ПВК як класів об'єктів. Наприклад, відомі

бібліотеки C++ налічують тисячі повторно використовуваних класів. Істотним відношенням класів об'єктів є відношення успадкування. Два класи об'єктів перебувають у відношенні успадкування, якщо один з них (спадкоємець) володіє всіма істотними ознаками другого (успадкованого) класу і має деякі додаткові властивості, поведінку або інтерфейс. За допомогою механізму успадкування природним чином локалізуються фіксована частина абстракції (представлена успадкованим класом, який називають суперкласом) і її змінна частина (представлена класом-спадкоємцем, який називають підкласом). Отже, внесення змін до ПВК або її настройка здійснюються як визначення потрібного підкласу, що спрощує супровід розробленої системи.

Суперкласи, які визначено не повністю і їх не можна самостійно використати, названо абстрактними класами. Для використання їх має бути додано підкласи, що успадковують всі властивості, характерні суперкласу, але мають додатково визначені деталі.

Прикладами абстрактних класів служать так звані контейнерні класи — структури зберігання даних, для кожного з яких визначено правила запам'ятовування або видачі чергового елемента контейнерного класу, однак при цьому реалізація цих операцій залежить від типів елементів, що заповнюють "контейнери". Поширеними прикладами контейнерів можуть бути списки, черги, стеки, матриці, таблиці. Механізм контейнерів реалізовано в C++ у формі так званих шаблонів (templates), створено спеціальні бібліотеки ПВК такого типу.

#### 12.3.4. Абстрактні архітектури програмних систем

Всяка архітектура визначається складом частин, з яких будується ціле, і способом їхньої композиції в ціле. Архітектура програмної системи визначається в термінах компонент, які виконують обчислення, та інтерфейсів між ними. Архітектура — це відображення правил декомпозиції складності проблеми.

Спектр задач проблемної галузі, що допускають схожі способи розв'язання їх, називають доменом предметної галузі. *Абстрактна архітектура* — це декомпозиція спільного вирішення для виділеного спектра завдань домену на підсистеми або ієрархію підсистем, на кожному рівні якої фіксуються можливі варіанти виділених параметрів та обмежень, котрим відповідають варіації складу виділених компонент.

Таким чином, абстрактні архітектури визначають глобальну структуру проектованої системи і функціональність її стабільних головних складових, що може розглядатись як фіксована частина **абстракції** підходу до вирішення виділеного спектра завдань.

Абстрактні архітектури звичайно будуються для вузьких проблемних галузей, що дозволяє досягти високої спільності підходів до вирішення властивих галузі завдань і, відповідно, високого рівня абстракції ПВК.

Прикладами ПВК абстрактних архітектур є:

- системи управління базами даних;
- архітектура компілятора, в яку можна вбудовувати різноманітні лексичні й синтаксичні аналізатори, генератори кодів;
- архітектура експертної системи, основаної на правилах, тощо.

**Спеціалізація** абстрактної архітектури здійснюється шляхом довизначення варіацій компонент. Наприклад, спосіб обчислення якогось показника в системі управління польотом літака може бути довизначено залежно від фізичних особливостей приладів, які встановлено на конкретному борту літака.

**Селекція** потрібної архітектури здійснюється шляхом аналітичного зіставлення характеристик проблемної галузі, що покривається абстрактною архітектурою, і характеристик тієї проблемної галузі, для якої створюється програмна система.

**Інтеграція** абстрактних архітектур у складні системи полегшується чіткою специфікацією їхніх інтерфейсів, завдяки якій кожна абстрактна архітектура може розглядатися, у свою чергу, як підсистема архітектури вищого рівня.

#### 12.3.5. Генератори прикладних застосувань

Генератори прикладних застосувань працюють подібно до компіляторів для мов програмування: вхідні специфікації автоматично трансклюються в потрібний об'єктний код (в тому числі в код мови програмування високого рівня). Особливість генераторів полягає в тому, що використовувані ними специфікації, як правило, є абстракціями дуже високого рівня для дуже вузьких доменів проблемних галузей, що дозволяє забезпечити повторне використання абстракцій як даних, так і процедур та архітектури проекту в цілому.

**Абстракція** в генераторі прикладних застосувань належить до родових знань про домен проблемної галузі і має для користувача вигляд специфічних для домену різноманітних моделей обчислень і даних.

Фіксована частина абстракції визначається тими складовими застосувань, які користувач не може змінювати в процесі генерації; змінна частина визначається тими параметрами, що задаються користувачем.

Комбінація фіксованої частини і діапазону змінних частин визначає спектр систем, які можна реалізувати за допомогою генератора. Ця міра дістала назву покриття домену генератором.

Розробник програмних систем працює винятково на дуже високому рівні абстракції. Усі деталі реалізації генератора є для користувача прихованою частиною абстракції, що позбавляє його необхідності (а також можливості) бачити, розуміти або модифікувати вихідний результат генератора.

**Селекція** або вибір генератора — це пошук такого генератора, чие покриття домену включає вимоги постановки потрібного завдання. До

нинішнього часу в обчислювальній практиці використовується лише невелика кількість дуже спеціалізованих генераторів з вузьким покриттям домену. Однак тенденція формалізації все нових і нових предметних галузей є такою, що можна очікувати появи великих бібліотек генераторів. Пошук у них генератора, відповідного потрібному завданню, має здійснюватися в термінах характеристик домену.

**Спеціалізація.** Розробник програмної системи спеціалізує генератор, задаючи її специфікації.

Зазначимо, що в тому разі, коли функціональні потреби системи, що розробляється, виходять за покриття домену генератора, його ревикористання стає неефективним.

**Інтеграція.** Як правило, генератор прикладних застосувань виробляє на виході завершено, готову для виконання систему, тому питання інтеграції для неї не ставиться. В тому разі, коли генератор генерує окремі компоненти систем, правила їхньої композиції визначаються в термінах абстракції високого рівня.

Залежно від вбудованих в реалізацію генератора способів спеціалізації може бути виділено різновиди генераторів, котрі набули широкого поширення. Назвемо їх.

**Генератори програм обробки даних у бізнесі.** Специфікації для таких генераторів визначають схеми баз даних, а їхню обробку визначають на так званих мовах четвертого покоління. Прикладами таких генераторів є:

- 1) управління даними — базується на одній моделі даних; наприклад, на реляційній або ієрархічній;
- 2) генератори форм документів — базуються на асоціативному пошуку даних (так звані генератори звітів);
- 3) графічні генератори звітів — аналогічні текстуальним, але на виході дозволяють отримувати графічне представлення результатів (даних) у формі графіків, діаграм тощо.

Спеціалізація може включати обмеження на окремі елементи даних та їхні відношення, обмеження на управління доступом і вироблення кута зору (view) на бази даних.

**Генератори експертних систем.** Експертні системи — це системи, які включають і застосовують для розв'язку задач домену експертні знання.

Абстракціями для них є узагальнені способи розв'язання задач у проблемних галузях шляхом обробки знань незалежно від їхньої природи, яка може суттєво відрізнятися, як, наприклад, при діагностиці людей і двигунів. Спосіб розв'язання задач є фіксованою частиною абстракції генератора експертних систем, а змінною частиною (за якою він спеціалізується) — експертні знання про конкретну область діагностики;

**Парсери й компілятори компіляторів.** Їхня основна абстракція — це регулярні вирази для генерації лексичних аналізаторів і кон-

текстно-незалежні граматики для генерації парсерів, що узагальнюють знання про принципи дії й алгоритми лексичних та інших аналізаторів, видавання помилок, генерацію коду тощо. Їхня спеціалізація відбувається за конкретними лексичними та граматичними правилами.

### 12.3.6. Абстрактні домени

Розвитком ідеї абстрактної архітектури є каталогізація domenів масового вжитку й узагальнення їхніх властивостей як поняття абстрактний домен [2]. Виконуючи аналіз подібності та розбіжностей сукупності domenів, експерти можуть помітити аналогії для таких, на перший погляд, відмінних речей, як продаж театральних квитків, резервування місць на рейси літаків або розподіл деталей на обробку для верстатів заводського конвеєра. Помітивши аналогії в структурі знань різних із синтаксичного погляду domenів, експерти можуть ввести **абстракції** об'єктів, котрі можна конкретизувати у відповідні об'єкти аналогічних domenів. Вимоги до систем, що належать до цих domenів, може бути представлено в термінах відповідного абстрактного домену.

Автори пропонують набір абстракцій проблем, які слугують базисом для генерації специфічних для конкретного домену сценаріїв збирання та перевірки вимог, використання їх під час експлуатації, складання вимог у формі юридичного контракту природною мовою. Підхід є першою систематичною спробою покриття простору (моделювання) інженерії вимог у сукупності domenів, котрі обслуговують менеджмент у бізнесі. Гіпотетично вони мають спиратися на ментальні моделі природних категорій когнітивних наук. Це є підхід, що обіцяє бути ефективним.

Простір моделей систем об'єктів становлять 13 базових ієрархій. Верхній рівень моделі системи об'єктів кожної ієрархії визначається з використанням базових: поведінки, складу об'єктів, агентів, структури домену для одного класу проблем. **Спеціалізація** кожної із цих моделей (у формі систематичного додавання різноманітних типів знань — станів, мети, подій) генерує простір до 200 листів-вузлів моделі систем об'єктів. Кожний вузол-лист визначається з використанням станів, переходів у стани, подій, об'єктів, агентів, структури домену, абстрактних відношень між об'єктами й агентами, передумов переходів у стани, властивостей та атрибутів об'єктів. Перелічимо 13 моделей верхнього рівня:

а) повернення ресурсів — двоспрямована передача ключових об'єктів від структури, що зберігає, до клієнтської і навпаки, наприклад, бібліотеки;

б) забезпечення ресурсами — односпрямована передача ресурсів від структури, що зберігає, до клієнтської та контроль за необхідністю поповнення ресурсами структури, що зберігає, наприклад складів;

в) використання ресурсів — односпрямована передача від структури, що зберігає, до клієнтської для обробки;

г) композиція елементів — агрегування із синтезом нового; наприклад, збирання деталей у продукт;

г') декомпозиція на елементи;

д) розміщення ресурсу — моделює зміни стану ресурсу щодо інших транзакцій, наприклад стани квиткових кас;

е) логістика — комплексна діяльність з планування, наприклад маршрутів парку машин;

е) спостереження за об'єктами — відстеження руху об'єкта в просторі чи у фізичній структурі, наприклад радарне;

ж) побудова сполучення між об'єктами — пов'язування об'єктів, які володіють інформацією, в топологію зв'язків;

з) управління агента об'єктом — представлення команд та керуючих компонент, до яких агенти посилають повідомлення, що викликають зміни в поведінці об'єктів, наприклад контроль суден у порту;

и) моделювання домену — комплексні засоби моделювання для агента-людини, наприклад військовика;

і) обробка ділянок — діалогові зміни стану ділянки, наприклад обробка слів або креслень;

ї) перегляд об'єкта — домени, в яких об'єкти не змінюють стану, але переглядаються агентами — людьми або комп'ютерами.

Перелік абстрактних доменів, безперечно, може бути поповнено, якщо певна спільнота професіоналів може узагальнити сукупність розв'язаних нею проблем у формі абстрактного домену. Прикладом домену масового використання, досвід роботи з яким дозволяє побудувати його абстракцію, є задачі так званої аналітичної обробки даних, чи не наймасовіші у сфері економічних завдань, зокрема АСУ, здавна відомі під назвою отримання розрізів даних [3, 4].

Для згаданих вище доменів можна навести чимало прикладів завдань, несхожих зовні, але вирішення яких вимагає маніпулювання об'єктами з аналогічною поведінкою. У межах цих доменів побудовано сотні працюючих програмних систем, і тому побудова для них класифікацій є проблемою скоріше організаційною, бо є достатня кількість експертів і достатня сукупність готових об'єктів для експертизи.

### 12.3.7. Патерни

Тип ПВК, який позначається терміном патерн, є принципово відмінним від тих, котрі розглянуто дотепер і котрі, до певної міри, належали до програмних модулів чи їх узагальнень.

*Патерни є абстракцією* спілкування (взаємодії) певної сукупності об'єктів у кооперативній діяльності, для якої визначено абстрактних учасників, їхні ролі, взаємовідносини та розподіл відповідальності.

Патерн фіксує певне узагальнене рішення, яке дозволяє спілкуватися розробникам та замовникам і розмірковувати щодо проблеми та її розв'язання. Це крок до стандартних інженерних шляхів застосуван-

ня успішних розв'язань до відомих проблем, як це ведеться у сталих інженерних дисциплінах (наприклад, для автомобіля відомо про певні набори сполучення вузлів, щоб одержати спортивний варіант, або висококомфортний, або для поганих доріг тощо). Інакше кажучи, це шлях до ревикористання корисного досвіду, який передається неформально за допомогою якісної, ментальної моделі опису знань.

Цей термін справді походить з архітектури. Патерн у контексті програмної інженерії є позначенням проблеми, яка виникає в багатьох контекстах, при цьому ядро її розв'язання можна представити так, щоб повторно використовувати його для більшості з них. Патерни описують окрему повторювану проблему проекту як визначену наперед схему її розв'язання шляхом специфікації абстракції її складових, їхніх ролей та взаємовідносин і відповідальності. Найчастіше патерни виражають фундаментальну парадигму структурування системи, як, наприклад, архітектури клієнт — сервер чи замовник — посередник — послуга.

Патерни мають певний контекст застосування; зазвичай можна визначити і мотиви доцільності застосування патерну, й аргументи, що суперечать їм. І кожне з рішень має певну вартість, а опис патерну має визначати цю вартість.

Відомо кілька схем опису патернів, у більшості з них визнається проблема, яку розв'язує взаємодія, контекст, в якому сформульовано проблему та підхід до її розв'язання. У наведеній схемі відображено не стільки технічні ідеї, скільки соціальні аспекти прийнятих рішень: проблема відображає мотиви розробки, контекст характеризує явища, реакцією на які є розв'язання проблеми, та аргументи проти розв'язання.

*Спеціалізація* патернів відбувається шляхом конкретизації (довизначення) об'єктів, котрі беруть участь у взаємодії.

*Селекція* (добір) патернів, адекватних системі, що будується, відбувається неформально, шляхом вивчення та зіставлення каталогів патернів.

*Інтеграція* обраних патернів у нову розробку відбувається неформально, найчастіше сукупність патернів для певного домену не інтегрується в мову проектування, а скоріше — це зібрання окремих одиниць, кожна з яких відповідає певній схемі взаємодії, котру можна конкретизувати для певних об'єктів розробки.

Відомо кілька таких зібрань, окремі з яких вже широко застосовуються в процесі розробки програмних систем, і не лише як певні готові рішення, які можна використати повторно, а і як засоби обговорення прийнятих рішень та визначення їх [5].

Об'єднання патернів у зібрання може відбуватися за класифікаційними ознаками, зокрема за сферою застосування їх. Так, патерни проектування можна умовно поділити на три групи.

**Креативні патерни** (від англ. create — створювати) відображають процеси створення екземплярів об'єктів. Нижче наведено приклади таких патернів:

- **ініціатор** — це відповідь на проблему, яка полягає в тому, що конкретний тип об'єкта часто змінюється; контекст визначається як багаторазова потреба породжувати екземпляри об'єкта; рішення — інтерфейс створення об'єктів відокремлюється від власне створення, яке вирішується шляхом визначення підкласу, котрий конкретизує, що саме створюється;

- **трансформер** — це відповідь на проблему організації перегляду файлів та трансформування їх у різні формати; рішення — відокремити механізм перегляду від алгоритмів створення нових форматів;

- **прототип** — це відповідь на проблему породжувати екземпляр об'єкта за заданим зразком; рішення — визначити тип за екземпляром прототипу і створити новий об'єкт як копію прототипу.

**Структурні або архітектурні патерни** визначають композиції класів об'єктів. Їхніми типовими прикладами є:

- **адаптор** — це відповідь на проблему пристосовування інтерфейсу до смаку персонального клієнта. Контекст — смаки часто змінюються. Рішення — визначити абстракцію інтерфейсу як суперклас, а настрійку на персональні вимоги клієнта — як підклас;

- **фасад** — це відповідь на проблему локалізації в підсистемах властивостей, що мають високу вірогідність змін. Контекст — найбільшу вірогідність змін мають об'єкти інтерфейсу. Рішення — передбачити можливі зміни в підсистемі та зосередити їх в одному класі;

- **пошарова структура** — це відповідь на проблему дотримування певного рівня абстракції в осмисленні проблеми на кожній стадії життєвого циклу розробки програмної системи (див. рис. 4.2.) Контекст — наявність розробників з різними рівнями кваліфікації, що зумовлює здатність виконувати роботу на відповідних рівнях абстракції. Рішення полягає в декомпозиції системи на ієрархію шарів, для кожного з яких встановлюється стандартизований інтерфейс між компонентами попереднього та наступного шарів. Кожен із шарів є сукупністю представлень віртуальних машин, відповідних певному рівневі абстракції, а компонента є реалізацією такої машини. Для кожного шару можна визначити певний спектр компонент, взаємозамінних та сумісних щодо їхніх інтерфейсів (засобів включення), але відмінних з погляду реалізації цих інтерфейсів (наприклад, різноманітні прилади в авіоніці можуть вимірювати ту саму величину).

Найчастіше спектр компонент — це перелічена множина компонент, і спектри можуть бути вкладеними один в одний. Кожна з компонент експортує свій інтерфейс до віртуальної машини верхнього шару й імпортує інтерфейс віртуальної машини нижнього шару, тобто трансформує перший у другий, але при цьому інкапсулює відображення. Спектри та компоненти можна визначити як складові

граматики, множина речень якої характеризує сімейство систем, котрі може бути породжено на базі патерну. При цьому може бути накладено певні допоміжні обмеження стосовно правил породження.

**Поведінкові патерни** визначають взаємодію класів та їхніх екземплярів (як розподіл абстрактних ролей) і їхню поведінку. Приклади:

- **ланцюг повноважень** — це відповідь на проблему роз'єднання відправників повідомлень від тих, хто їх обробляє. Контекст — потреба в роздільному функціонуванні перших і других. Рішення — повідомлення передається від об'єкта до об'єкта, доки не дійде до того, хто виконає його обробку;

- **клієнт/сервер/сервіс** — це відповідь на проблему інкапсулювання подробиць реалізації програмних послуг, щоб вивільнити клієнтів від необхідності знати їх (згідно з принципом приховування інформації, про який ішлося в главі 3). Контекст — сервіс може бути запрошено паралельно й одночасно багатьма клієнтами. Рішення — побудова двох класів. Перший з них — сервіс — інкапсулює послідовність виконання послуг, підтримує стани виконання послуги і передає події, котрі виникають у процесі виконання послуги. Другий — сервер — інкапсулює ресурси та послуги сервісів і визначає абстрактний інтерфейс з клієнтом, конкретизація якого ініціює конкретне виконання конкретного сервісу.

Підсумовуючи зазначене вище, можемо сказати, що патерни є спробою висловити інтуїтивні рішення, набуті досвідом окремих провідних спеціалістів, як формалізовані абстракції, конкретизацію яких можна виконати широким колом спеціалістів за умов, відповідних їхнім вимогам.

Зазначимо, що наведені приклади можуть декому здатися давно знайомими, і так воно і є, але новим є представлення їх у вигляді чітко визначених правил взаємодії об'єктів.

### 12.3.8. Каркаси (Frameworks)

**Каркас проблеми** визначає готову структуру головних складових частин, на які має поділитися система, побудована для розв'язання проблеми. ПВК типу каркас [6] виникли як розвиток об'єктно-орієнтованого підходу до розробки програмних систем [16].

Згідно з парадигмою згаданого підходу каркас об'єднує множину класів об'єктів, які взаємодіють між собою для досягнення множини цілей, котрі розв'язують проблему. Водночас каркас є високорівневою **абстракцією** проекту реалізації програмної системи.

За способом **спеціалізації** каркаси поділяють на два різновиди. **Каркас типу "біла скринька"** має у своєму складі абстрактні класи, які неповно визначають мету та інтерфейси компонент. Щоб трансформувати такий каркас у конкретну прикладну систему, достатньо породити шляхом наслідування конкретні класи й доповнити при цьому визначення відповідних методів. Але цей процес потребує знання про внутрішню побудову каркаса, що суттєво утруднює його використання.

*Каркас типу “чорна скринька”* — це структура, окремі вузли якої означено як такі, визначення котрих відкладено (так звані “*гарячі плями*”). На відміну від попереднього типу каркаса, для заповнення “гарячих плям” пропонується спектр можливих альтернативних класів — претендентів зайняти місце у “гарячій плямі”. Кожна пляма відповідає одному з аспектів змінюваності системи. Вибираючи один з альтернативних класів, ми отримуємо реалізацію відповідного йому варіанта системи.

Отже, в проектуванні системи передбачається її здатність змінюватися за окремими аспектами, фіксуються позиції можливих “збурень” у конструкції системи і передбачаються можливі варіанти. Створення системи виглядає як визначення конкретної конфігурації компонент, відповідних “гарячим плямам”. Визначення конкретного варіанта системи проводиться як вибір “з полиці” потрібних компонент.

При цьому від конструктора конкретного варіанта вимагається лише знати перелік “гарячих плям”, суть аспектів, варіантність яких визначається цими “плямами”, і суть доступних варіантів за кожним із згаданих аспектів. Проектування системи у формі каркаса потребує від її конструктора спрямувати увагу на майбутні зміни, передбачити перелік таких змін, забезпечити максимальну локалізацію ефекту від втілення їх і в такий спосіб продовжити термін життя системи. Мабуть тому вживання каркасів стало поширюватись у новітніх технологіях програмної інженерії.

**Селекція** (добір) каркасів відбувається як неформальний, інтуїтивний процес зіставлення можливостей каркаса з вимогами цільової розробки.

**Інтеграція** каркасів можлива тільки шляхом включення їх як підсистем у цільові розробки.

Прикладом каркаса є система банківських розрахунків, для якої визначено дві “гарячі плями”: компоненту ідентифікації рахунка клієнта (зазвичай у кожній країні діють відповідні стандарти) та компонента реакції на прохання клієнта зняти певну суму з його рахунка, якщо така сума перевищує наявну суму рахунка (окремі банки можуть у такому разі кредитувати клієнтів певних категорій або на певних умовах).

## 12.4. Створення повторно використовуваних компонент

### 12.4.1. Вияв потенційних ПВК

Раніше у п. 12.1 було зазначено, що тільки систематичне повторне використання має перспективи стати ефективним засобом творення програм, тож у цьому розділі буде йти мова про компоненти як результат діяльності, спеціально спрямованої на можливість використання їх у багатьох прикладних застосуваннях.

Будемо називати групу подібних програмних продуктів, які належать до одного сегмента ринку (конкурують на ньому або доповнюють одне одного) *лінією продуктів* [7].

Лінія продуктів є найбільш перспективною нішею використання ПВК. Поряд з нею ПВК можуть використовуватись у кількох підсистемах одного прикладного застосування, у різноцільових застосуваннях, між якими можна виявити певні аналогії, як в абстрактних доменах (див. п. 12.3.6.).

Як засвідчують експерти, процес систематичного повторного використання може бути започатковано тільки після того, як буде створено певну кількість конкретних систем програмування з наведених вище різновидів, тобто буде напрацьовано базу для узагальнення (програмістські “легенди” показують, що ця кількість дорівнює в середньому трьом). За цих умов стандартний шлях виявлення ПВК становлять такі етапи:

- аналіз спільності та розбіжностей у створених досі системах;
- зіставлення вимог до нової системи з можливостями створених;
- прогнозування можливих змін у вимогах (до тих систем, що аналізуються) та виникнення потреб у нових системах (див. главу 8);
- пошук компонент — кандидатів для повторного використання.

У попередніх розділах ми представили процес розробки програмних систем як послідовне створення низки моделей (аналізу вимог, проекту, реалізації, тестування), сукупність яких дозволяє трансформувати побажання замовника в працюючу програмну систему.

Готова система постає перед тим, хто має зрозуміти й проаналізувати її, як послідовність формалізованих моделей у відповідних нотаціях, властивих тому чи іншому методу (як-от подані у главах 3—5 діаграми UML, І. Джекобсона чи С. Шлеєр та С. Меллора), а також як неформальні описи, коментарі до моделей та рекомендацій із застосування для користувачів.

Кожний з елементів згаданих моделей може бути потенційним кандидатом на повторне використання — актор, сценарій, інтерфейси між ними, об’єкти сценарію та відношення між ними, об’єкти, класи та підсистеми проекту, об’єкти та класи реалізації, діаграми переходів у стани та їхні фрагменти тощо. Перетворення названих елементів у ПВК відбувається шляхом узагальнення й перетворення їх на абстрактних акторів, абстрактні сценарії, абстрактні класи, підсистеми тощо.

При подальшому проектуванні абстрактні сценарії та підсистеми можуть втілюватися в такі категорії ПВК, як генератори, каркаси, класи об’єктів, тоді як більш детальні елементи моделей (класи, інтерфейси тощо) частіше відображаються в компоненти вихідного коду (див. п. 12.3.). Ясно, що чим раніше на етапах життєвого циклу буде виявлено кандидата у ПВК, тим меншою буде відповідна йому когнітивна відстань (див. п. 12.3.), тим більший виграш від ПВК можна чекати в майбутньому.

Побудова певної абстракції має на меті розширення спектра її придатності шляхом реалізації узагальненого розв'язання проблеми і забезпечення засобів подальшої спеціалізації абстрактного елемента на конкретний конкретний випадок застосування ПВК. Інакше кажучи, ми вважаємо кандидатами у ПВК такі елементи, для яких можна прогнозувати змінність (варіантність) певних властивостей і передбачити спектр можливих змін, адаптація до яких якраз і створює можливість повторного використання. Тому пошук кандидатів у ПВК є не лише визначенням необхідних абстракцій, а також і визначенням точок і характеру прогнозованих змін та відповідних механізмів адаптації до них.

Можна назвати властивості проекту, для яких доцільно зафіксувати варіантність сценарію та акторів [8]. Це такі властивості.

**Варіантність інтерфейсів для категорій користувачів або для програмних систем.** Приклади:

- 1) керівники різних рівнів потребують різнорівневих подробиць для видачі показників;
- 2) прикладні застосування відрізняються тим, що використовують різні СУБД;
- 3) різні інсталяції потрібні для окремих підрозділів організації, котрі працюють на різних конфігураціях устаткування.

**Робота з кількома взаємозамінними типами об'єктів-сутностей.**

Приклади:

- 1) Різноманітні типи приладів, котрі можуть вимірювати або регулювати однакові показники на конкретному “борту” літака в різних системах вимірювання, як-от відстань у кілометрах чи милях, пальне у літрах чи галонах тощо;
- 2) матриці з різною щільністю заповнення елементів (діагональні, стрічкові, розріджені тощо).

**Функції, виконання яких в окремих випадках необов'язкове.**

Приклади:

- 1) клієнт, відправивши поштою оплату покупки, може інколи обумовити зворотне повідомлення про надходження платні;
- 2) під час тестування можна видавати чи не видавати контрольні значення;

- 3) дані в процесі введення можна контролювати чи не контролювати.

**Обмеження на застосування або на правила бізнесу.** Приклади:

- 1) в Україні правила обчислення податків часто змінюються;
- 2) правила перевірки прав доступу клієнта до банківського рахунка постійно вдосконалюються;
- 3) обчислення зарплати має свої особливості в окремих організаціях, а вимоги контролюючих органів до відповідної звітності також є змінним чинником.

**Усунення помилок.** Приклади:

- 1) коригування на основі гранично допустимих даних;

- 2) перехід до стану, що передував тому, в якому знайдено помилку функціонування;

**Гнучке налаштування на масштаби або швидкість обробки, тобто необхідність регулювання співвідношення пам'яті/швидкість.** Прикладом може бути ситуація, коли час очікування відповіді на запит до системи визначається залежно від посади того, хто запитує, тобто відрізняється для керівників різних рівнів (чим вищий начальник, тим менше він згоден чекати відповіді на свій запит, а прискорення реакції системи здебільшого досягається за рахунок збільшення ресурсу пам'яті).

#### 12.4.2. Специфікація варіантності вимог

Дослідивши елементи моделей готових систем, які було розроблено раніше, та визначивши потенційних кандидатів на роль ПВК, позначимо тепер конкретні місця можливих змін у наявних представленнях моделей кандидатів та способи адаптації згаданих моделей до таких змін.

Будемо називати *точками варіантності* текстуально виділені позиції в представленні моделей компоненти, для яких прогнозується можливість змін і передбачаються засоби адаптації до них.

Точки варіантності позначаються на моделях. Точка в овалі сценарію позначає прогнозовану змінність певних властивостей або функцій сценарію.

Приклад. На рис. 12.1 представлено об'єкт “Банківський рахунок” для деякої банківської системи. Система ідентифікації рахунків банку в кожній країні має свої особливості — в Японії вона відрізняється від прийнятої в Швейцарії, в Польщі — свої особливості тощо.

Точка в прямокутнику класу — змінність його атрибутів або операцій тощо.

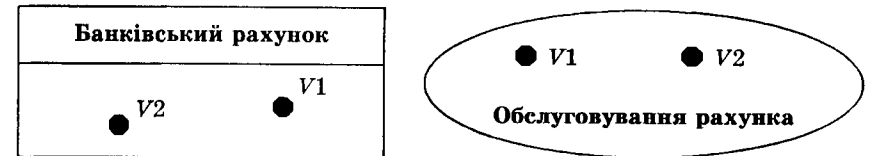


Рис. 12.1. Представлення варіантів на елементах моделей

Точкою варіантності *V 1* на малюнку позначено прийняту для даної прикладної системи стратегію перевірки правильності вказаного ідентифікатора розрахункового рахунка, варіант якої залежить від того, в якій країні використовується система. Об'єкт “Банківський рахунок” має дві точки варіантності — *V 1* і *V 2*, друга з яких позначає ту обставину, що прийнята стратегія поведінки банку підлягає зміні, якщо клієнт, вичерпавши свій кредит, намагається зняти гроші з рахунка.

Для певних видів змінності широко застосовуються так звані патерни як штампи забезпечення локальності змін (див. п. 12.3.). Нагадаємо, що терміном патерн позначаються стандартні об'єктно-орієнтовані рішення для стандартних проблем, описані стандартним способом. Патерни фіксують переважно перевірені часом штампи взаємодії об'єктів. Відзначимо низку таких штампів із числа наведених у джерелі [5], які корисні для деталізації сценаріїв у побудові ПВК:

— відокремлення інтерфейсу від обробки робить систему стійкою до змін інтерфейсу як найбільш динамічного аспекту вимог. Кілька можливих варіантів інтерфейсу представляються за допомогою патерну, названого *обсервером*, за яким інтерфейс уявляється як абстрактний об'єкт та сукупність об'єктів, котрі його успадковують (рис. 12.2).

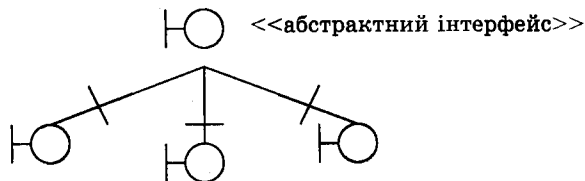


Рис. 12.2. Варіантність інтерфейсів

— варіантність використовуваних баз даних можна представити за допомогою патерну, названого *мостом*, за яким абстракція об'єкта (узагальнення доступу до баз даних) відокремлюється від своєї реалізації (звернення до конкретної СУБД);

— розширення функції може бути представлене патерном декоратора, за яким складний сценарій розбивається на сукупність простіших, кожний з яких є розширенням функції абстрактного об'єкта, названого декоратором (приклад об'єкта-декоратора подано на рис. 12.3).



Рис. 12.3. Варіантність функцій

— якщо до системи входять кілька досить великих підсистем, котрі мають велику кількість класів об'єктів, доцільно використати патерн *фасаду*, за яким інтерфейс з підсистемою доцільно зосередити в спеціальному класі. Поняття фасаду відіграє велику роль у процесі повторного використання, тому зупинимося на ньому докладніше.

Звичайно, підсистему можна розглядати як систему компонент з узгодженою взаємодією, які можуть спільно функціонувати, успадковувати одне одного. Для користувача така система виглядає як певний комплекс послуг, опис яких власне і подається як фасад.

Ми кажемо, що прикладна система ревикористовує готові компоненти і системи компонент шляхом імпорту їхніх властивостей. Нагадаємо, що властивості, які імпортуються, становлять видимі частину абстракції.

При цьому для кожного випадку ревикористання ПВК звичайно імпортує, а прикладна система, відповідно, експортує тільки потрібну в даному разі частину властивостей, якими володіє ПВК. Таким чином, співвідношення видимої і прихованої частин абстракції може змінюватися для кожного випадку ревикористання.

Видима для даного випадку сукупність властивостей представляється власне фасадом. Фасад являє собою механізм доступу до тих властивостей системи компонент, які потрібні для конкретного ревикористання. Наприклад, система компонент, що реалізує доступ до баз даних, може мати фасадом чотири стандартних операції — “читати,” “писати,” “створити,” “знищити”. Деякі важливі операції, такі, як перевірка правильності при введенні, відновлення, підтримка мультидоступу можуть, залежно від використання, належати до прихованої частини або бути виведеними на фасад. Інші внутрішні властивості баз даних може бути інкапсульовано.

На фасад може бути виведено будь-які знання про ПВК, наприклад, її сценарій, інструкції з використання тощо. Фасад — це погляд на систему компонент того, хто її використовує. Побудовано фасад так, щоб втілити відомий принцип Парнаса “приховування непотрібних знань”.

Одна система компонент може мати кілька фасадів — якість фасаду визначає успіх ревикористання. Фасад має бути стабільним, а те, що в нього не ввійшло, може змінюватися (наприклад, використання конкретного серверу СУБД). Приклад фасаду подано на рис. 12.4 в нотатії UML.

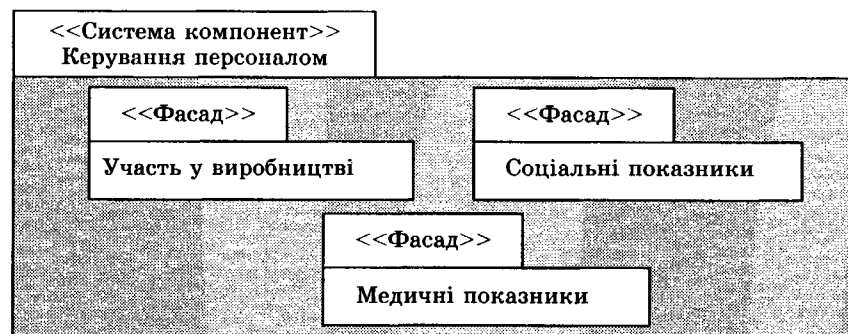


Рис. 12.4. Приклад фасаду.

#### 12.4.3. Конкретизація варіантності вимог

Конкретизація варіантної точки може здійснюватися як вибір однієї з визначених наперед можливостей (як, наприклад, згадувана вище система ідентифікації банківського рахунка) або як довизначення користувачем адаптованого до конкретного застосування екземпляра ПВК. Звичайно, фасад системи містить відомості про наявність варіантних точок і засобів їхнього довизначення.

Будемо називати *вирішенням варіантної точки* ПВК механізм визначення конкретного варіанта використання ПВК та її адаптації до такого використання.

Можна навести типові механізми вирішення варіантної точки ПВК, тобто її адаптації або настроювання.

**Механізм успадкування.** Окремі властивості поведінки суперкласу конкретизуються в підкласі. Точний зміст цього процесу значною мірою визначається конкретною об'єктно-орієнтованою мовою програмування. На рівні стадії аналізу вимог вирішення варіантності шляхом наслідування позначається на моделі сценаріїв відношеннями “розширює” та “використовує”.

Відношення розширення сценарію (див. п. 3.3) доцільно застосовувати, якщо функції та поведінка одного сценарію (суперкласу) доповнюється функціями та поведінкою іншого (підкласу). Це розширення може бути представлено явною варіантною точкою (наприклад, якщо деталізацію певного рішення відкладено на наступну фазу життєвого циклу). Або неявною, коли посилаються на окремий документ, в якому наводяться правила розширення.

Звичайно, сценарій уточнюється за кілька ітерацій, на кожній з них визначений раніше сценарій може бути деталізовано і доповнено новими функціями. Таке доповнення зручно відобразити в моделі сценаріїв відношенням розширення. Розширення, по можливості, не повинно змінювати розширюваний сценарій, тоді досягається стабільність сценарію-суперкласу, бо він не залежить від своїх розширень.

Відношення використання доцільно застосовувати, якщо один сценарій включається повністю (без будь-якого настроювання) до іншого. Важливо розрізняти кілька самостійних сценаріїв та кілька варіантів того самого сценарію.

**Механізм конфігурації.** Механізм конфігурації застосовується тоді, коли точка варіантності позначає можливий вибір із сукупності передбачених наперед взаємозамінних компонент (у тому числі на рівні сценаріїв). Приклад вирішення варіантності шляхом конфігурації наведено на рис. 12.1, де показано об'єкт “Банківський рахунок” з двома варіантними точками, для кожної з яких передбачено вибір однієї з передбачених альтернативних можливостей, що зумовлено конкретними обставинами використання об'єкта.

Механізм конфігурації, як і успадкування, безпосередньо використовується у ПВК, що належать до категорії каркасів (див. п. 12.3.8.)

Включення тих чи інших компонент або систем компонент (елементів конфігурації) в загальну композицію прикладної системи для складних великих ПВК може породити велику кількість версій такої композиції, залежно від конкретних обставин застосування. Тому управління конфігурацією розглядається як окрема гілка в дереві знань програмної інженерії, і ступінь автоматизації цього процесу вважається навіть як один з показників оцінки технологічної зрілості організації розробників. Більш детально ці питання розглядаються в главі 10.

**Механізм настроювання за параметрами.** Варіантна точка може означати, що вирішення варіантності передбачено виконувати шляхом задання параметрів настройки. В сучасних системах програмної інженерії настроювання за параметрами може бути також достатньо складним. Для конструкції процедур у класичних мовах програмування настроювання за параметрами потребувало заміни формального значення параметра на вказане фактичне, причому всюди, де було текстуально локалізовано формальне значення, тобто реалізація зводилася до пересилання фактичних значень на поля пам'яті, відведені для формальних параметрів. У сучасних засобах створення програмних систем використовується цілий спектр достатньо складних типів настроювання за параметрами.

Серед них відзначимо такі:

- 1) параметром є значення змінної або список значень. Настроювання здійснюється шляхом встановлення відповідних значень змінних;
- 2) параметром є ім'я або список імен. Настроювання здійснюється шляхом підстановки імені як фактичного значення параметру замість входження формального параметра до вихідного тексту програми;
- 3) параметром є явна умова прийняття рішення або умова обмеження. Настроювання здійснюється шляхом підстановки умовного виразу замість входження формального параметра до вхідного тексту програми;
- 4) параметром є неявна умова прийняття рішення або умова обмеження, тлумачення якої визначається станом бази знань. Настроювання вимагає виведення явної умови на підставі бази знань і наступної підстановки як у випадку 3;
- 5) параметром є вираз, який визначається шляхом генерації. Настроювання потребує спеціального генератора;
- 6) параметром є організація носіїв даних. Настроювання вимагає виклику відповідних серверів (серверів баз даних, файлів тощо);
- 7) параметрами є імена послуг та функцій, що використовуються. Настроювання здійснюється шляхом звернень до відповідних ПВК.
- 8) параметром є специфікація серверу, що використовується. Настроювання здійснюється шляхом реалізації серверу;
- 9) параметром є специфікація інтерфейсу. Настроювання здійснюється шляхом використання інструментальних засобів конструювання інтерфейсу;

10) параметрами є показники середовища виконання. Настроювання здійснюється шляхом виконання серверів інсталяції;

11) параметрами є необхідні виміри під час виконання ПВК. Настроювання здійснюється шляхом генерації коду;

12) параметрами є показники якості виконання (швидкодія, надійність тощо). Настроювання здійснюється шляхом генерації коду;

13) параметром є клас об'єкта, що використовується в родовому класі (в шаблоні або контейнерному класі). Настроювання здійснюється спеціальними засобами мови програмування;

14) параметром є певне поняття проблемної галузі. Настроювання здійснюється шляхом виведення його тлумачення з бази знань.

Перелік можливих випадків вирішення варіантності за допомогою визначення параметрів постійно нарощується з появою нових засобів генерації коду.

Коли ми розв'язуємо питання про доцільність використання системи компонент, ми зіставляємо її варіантні точки з тими, яких потребує створювана нами система. Якщо вони збігаються, ми імпортуємо всю систему компонент, як на рис. 12.5. Якщо ні — шукаємо інших кандидатів на використання.

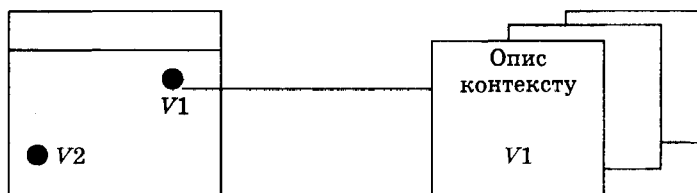


Рис. 12.5. Випадок використання системи компонент

Якщо на рівні прикладної системи ми фіксуємо лише один варіант конфігурації, то здійснюємо спеціалізацію відразу, бо підтримка варіантності в проекті завжди потребує ресурсів пам'яті і часу, а якщо варіантів не вистачає, додаємо нові, як на рис. 12.6.

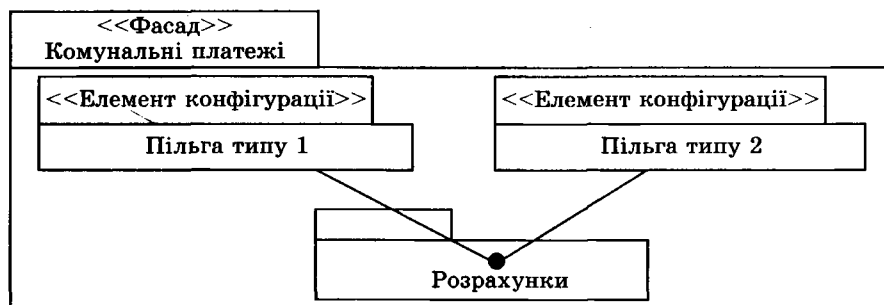


Рис. 12.6. Варіанти конфігурації

Коли ми створюємо систему компонент для повторного використання, то, як ми вже зазначали, умовою успіху є можливість потенційного користувача швидко зрозуміти, яку з його проблем покриває дана система компонент. Документування ПВК за допомогою фасаду та варіантних точок з визначеним механізмом вирішення їх дає лаконічне уявлення про властивості ПВК, визначальні для її використання. Інакше кажучи, для того щоб використати систему компонент, ми маємо якомога раніше врахувати всі можливості, які вона надає, тобто розв'язати проблему її використання на стадіях аналізу та проектування, формулюючи сценарії використання нової системи в термінах фасаду системи компонент (його об'єктів, точок варіювання тощо).

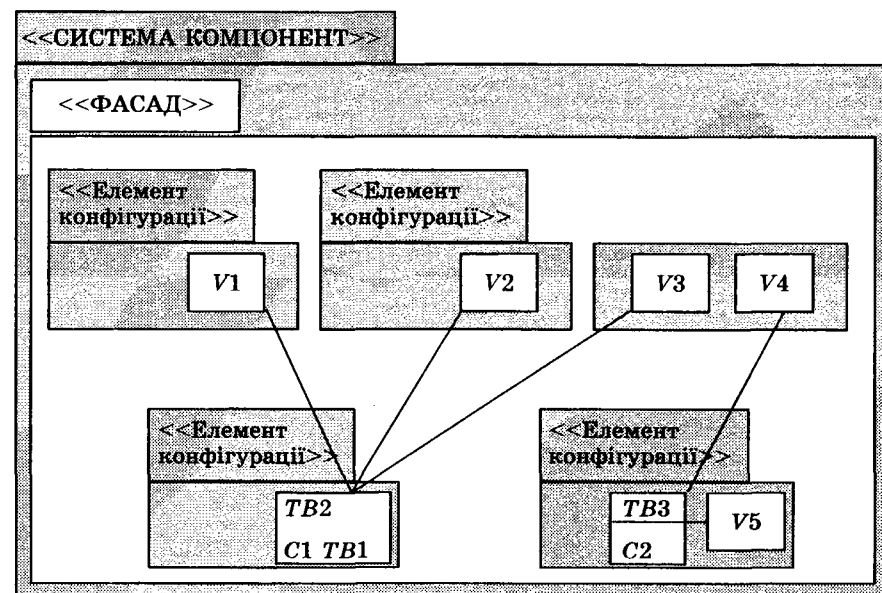


Рис. 12.7.

Рис. 12.7 ілюструє цю тезу. Видима частина — не заштрихована, біла.

Приклад. Новий банківський консорціум, використовуючи можливості Internet, організовує “банк вдома”, — тобто електронний банк, який обслуговує фінансування маркетингу, умови котрого швидко змінюються, тому реалізується не монолітна структура, а система як композиція компонент, що допускають адаптацію. Можлива розподілена архітектура клієнт-сервер або система рівноправних компонент. Клієнтські

компоненти працюють на персональних машинах або через Internet. Малі оплати можуть здійснюватися через Internet. “Банки вдома” мають спілкуватися з іншими фінансовими об’єктами, можливо, програмне забезпечення для “банку вдома” може мати спільні ПВК з кредитними, страховими, телефонними та іншими компаніями.

Понятійна база для цього домену включає цілу низку понять, що відображають типово банківські операції (“зареєструвати”, “вкласти”, “зняти з рахунку”, “перевести на інший рахунок”) та об’єкти, з якими вони оперують (позичка, кредит, акції, вклад, закладна, накладна, страховка, вексель, портфель цінних паперів, гроші готівкою, електронні гроші, чек, відсотки тощо). Тоді рис. 12.7 може бути ілюстрацією до цього прикладу, якщо вважати:

- V1 позначає елемент конфігурації, який здійснює блокування видачі, якщо рахунок вичерпано;
- V2 позначає елемент конфігурації, який здійснює підвищення відсоткової ставки, якщо кредит перевищує залишки рахунку;
- V3 позначає елемент конфігурації, який здійснює застосування гнучкої стратегії кредитування з урахуванням ризику;
- C1 позначає елемент конфігурації, який здійснює ідентифікацію рахунку клієнта;
- TB1 є варіантною точкою, яка визначає середовище клієнтських компонент;
- TB2 є варіантною точкою, яка визначає поведінку банку, якщо замовлена виплата перевищує залишки на рахунку;
- TB3 є варіантною точкою для ідентифікації особистого рахунку клієнта;
- C2 є модуль контролю ризику.

Незаштрихована частина рисунка відповідає видимій інформації.

#### 12.4.4. Проектування ПВК

Базою для пошуку компонент і систем компонент є сукупність напрацьованих прикладних систем та готових систем компонент. Означена сукупність розглядається як єдина система, яку будемо називати *суперсистемою*. До її фасаду включається тільки та інформація, яка стосується використання кожної із складових суперсистеми.

Домовимось, що суперпозиція акторів і сценаріїв складових суперсистеми є першим наближенням акторів та сценаріїв власне суперсистеми (рис. 12.8).

Наступним кроком є узагальнення включених сценаріїв та акторів і побудова абстрактних сценаріїв та акторів суперсистеми в такий спосіб, щоб сценарії та акторів кожної із складових систем було поділено на дві категорії — на ту, що є конкретизацією сценарію

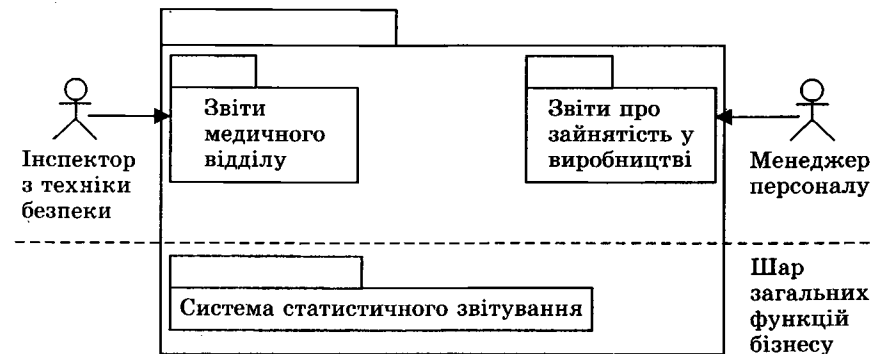


Рис. 12.8. Використання суперсистеми

чи актора суперсистеми шляхом вирішення точки варіантності, та на ту, що є індивідуальною властивістю конкретної складової суперсистеми і не використовуються в інших складових (рис.12.9).

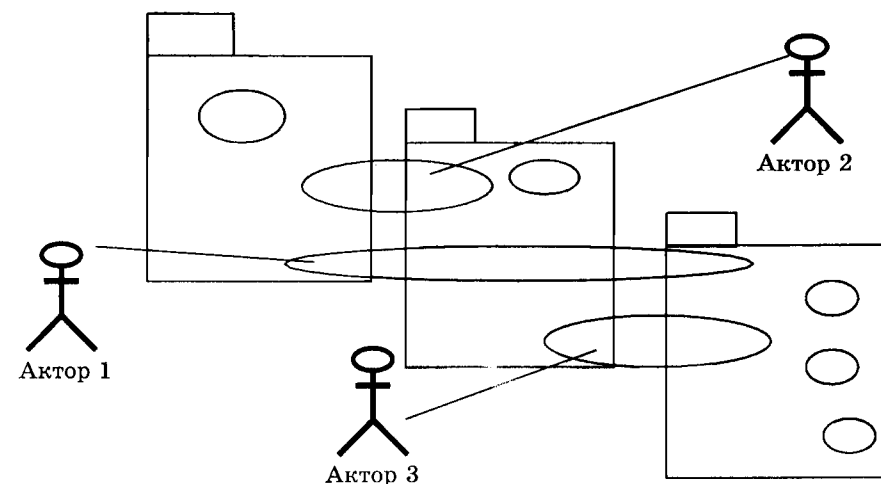


Рис. 12.9. Приклад розташування сценаріїв у суперсистемі

Перші з них є кандидатами у ПВК. Щоб перетворити їх на справжні ПВК, потрібно вдосконалити їхні інтерфейси, можливо, більш чітко визначити їхні фасади і правила та обмеження застосування. Якщо не вдається абстрагувати сценарій у цілому (тобто виділити ПВК-кандидатів на стадії аналізу вимог до систем), проводимо дослідження моделей проектів систем, включених нами до складу суперсистеми з

метою виявлення тих її елементів, які можуть вважатися кандидатами у ПВК (абстракції класів, моделей переходів у стани тощо).

У процесі побудови моделі проекту суперсистеми визначаються й уточнюються такі аспекти:

- взаємодія прикладних систем, охоплених суперсистемою;
- класи ПВК-проекткування, типи підсистем, утворені з них системи компонент;
- фасади систем компонент та ті їхні властивості, які імпортуються за допомогою фасадів;
- розподіл функцій сценаріїв, визначених для моделі суперсистеми, між охопленими нею прикладними системами та системами компонент.

Треба зауважити, що на першій стадії архітектурної роботи компоненти і фасади не може бути визначено одразу. Вони уточнюються разом із стабілізацією архітектури.

Модель проекту суперсистеми визначається щодо шарів, наведених на рис. 4.2, кожна підсистема належить одному шару і залежить тільки від компонент цього шару або шару, який розташовано нижче.

Наступний крок — виявлення загальнозначущих акторів. Якщо суперсистема взаємодіє з певними акторами, то конкретні прикладні застосування та підсистеми, для яких вона є узагальненням, мають також взаємодіяти з ними. Підсистеми нижнього рівня, які імпортує система, не експортують своїх акторів, бо актор — це зовнішня відносно до системи сутність, а імпортовані властивості є внутрішніми відносно до підсистем, які їх використовують.

Коли ми говорили про складові суперсистеми, то мали на увазі системи, які на сучасному рівні (з побудовою всіх необхідних моделей аналізу, проекту тощо) реалізують окремі прикладні застосування та відповідні системи компонент. Але, окрім таких складових, для побудови суперсистеми часто необхідно враховувати так звані успадковані системи.

*Успадкованою системою* називають діючу систему, створену застарілими (небажаними або навіть невідомими для команди розробників) засобами й технологіями проектування, яку, однак, усе одно експлуатують для управління значущими даними бізнесу і підтримки процесів бізнесу, бо вона задовільно виконує свої функції, а її модифікація потребує чималих коштів.

Прикладом може бути база даних великого обсягу, зібрана за допомогою застарілої СУБД, дані якої все ще потрібні.

Суперсистема, яка використовується як узагальнення спектра завдань домену, має враховувати і ті завдання, які представлено успадкованими системами. Для досягнення цієї мети почнемо з того, що побудуємо оболонку, яка інкапсулює послуги успадкованої системи й забезпечує взаємодію з нею в термінах сучасних моделей розробки систем.

Така оболонка виконує роль транслятора інтерфейсів успадкованої системи в інтерфейси інших складових суперсистеми. Тоді ця

оболонка розглядається як система компонент і вбудовується в суперсистему. Для неї будується фасад, через який експортуються її послуги на всіх фазах життєвого циклу нових систем, котрі взаємодіють з успадкованою системою. Нові системи, які її використовують, розглядають її як підсистему з віддаленими об'єктами, що визначаються оболонкою і якими нова система маніпулює шляхом звернення до віддаленої підсистеми. Для однієї успадкованої системи може бути побудовано кілька оболонок і кілька фасадів. На рис. 12.10 подано приклад використання успадкованої системи розрахунків, зробленої мовою програмування Кобол.

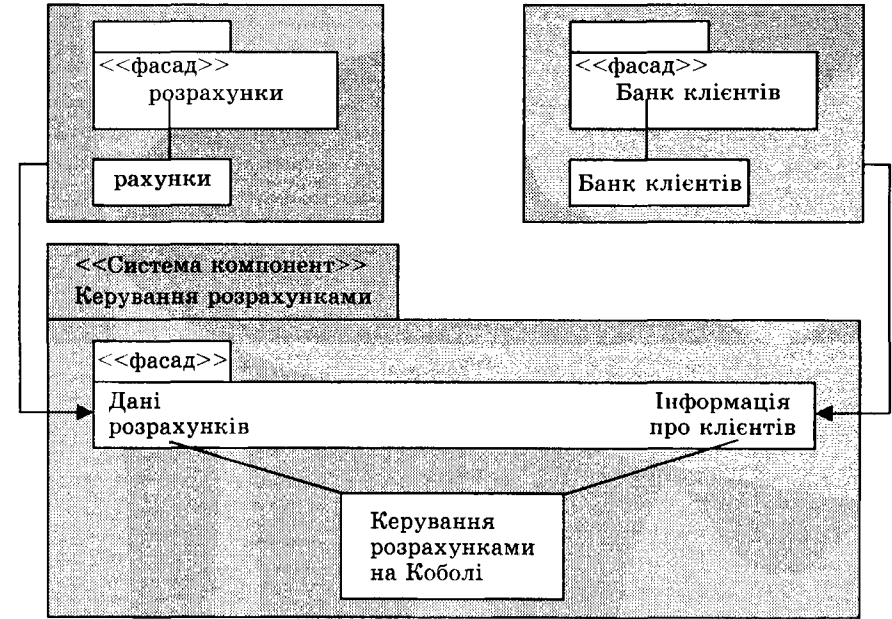


Рис. 12.10. Приклад використання успадкованої системи

## 12.5. Інформаційне забезпечення повторного використання

Якими б корисними не були створені ПВК, їх можуть знайти їхні потенційні користувачі тільки при умові, що хтось поінформує їх про наявність та властивості ПВК (див. п. 12.1).

Назвемо апарат інформування потенційних користувачів про наявність, властивості та умови використання ПВК *інформаційним забезпеченням повторного використання*. Його основне завдання — навігація користувачів у просторі колекцій ПВК, котрі розрізняються за призначенням, категоріями (див. п. 12.3) та представленням,

належністю до різних доменів (проблемної галузі). Мета такої навігації — знайти готові рішення, що відповідають вимогам цільової системи, яку має створити користувач.

Згадаємо, як організовано інформаційне обслуговування відвідувачів бібліотеки. Кожний примірник книги (об'єкта зберігання) має свою інформаційну картку в каталозі бібліотеки, котра є інформаційною моделлю книги і виконує роль так званого пошукового образу — моделлю відомостей, за якими інформаційна система виконує обслуговування, зіставляючи поданий читачем так званий пошуковий запит (відомості про об'єкти, які він шукає) з тими пошуковими образами, що відповідають діючим об'єктам зберігання. Так само для ПВК як об'єктів зберігання має бути побудовано відповідні каталоги.

Успіх пошуку звичайно забезпечується якістю каталогів, зокрема інформаційною моделлю, на якій їх побудовано.

Інформаційна модель ПВК, за допомогою якої має бути представлено інформаційний образ окремої ПВК у каталозі (пошуковий образ ПВК), орієнтована на розуміння людиною функцій ПВК (інколи досить складних) та на можливість зіставлення їх з власними потребами. Очевидно, таке завдання набагато складніше від бібліотечного пошуку, коли в ролі інформаційної моделі звичайно виступає ім'я автора, назва або жанр книжки тощо.

Можна назвати кілька відомих підходів до побудови інформаційної моделі ПВК [9, 10]:

- пошуковим образом ПВК може бути список ключових слів, які найчастіше згадуються в тексті ПВК. Це так званий спосіб вільного індексування;

- пошуковий образ ПВК може бути створено на базі заздалегідь побудованої онтології домену проблемної галузі (див. п. 3.2.1.). Це спосіб ієрархічної класифікації, прикладом якого є універсальна десятикова класифікація — УДК, що використовується, зокрема, для каталогізації наукових публікацій: окремим галузям знань присвоюються унікальні номери, так само робиться для їхніх підгалузей до певної глибини, між якими існує відношення рід — вид. Тоді кожна публікація в науковому виданні маркується (кажуть, індексуються) послідовністю номерів, розділених крапками, що позначають відповідні галузь, підгалузь, розділ тощо і до тематики яких належить дана публікація. Наприклад, індекс УДК 681.3 позначає розділ кібернетики “системи програмування”, і якщо хтось хоче знайти наукові публікації з даної теми, він може вказати індекс УДК у пошуковому запиті, якщо в бібліотеці є тематичний каталог, пошукові образи котрого мають у своєму складі індекс УДК. УДК є найпоширенішою міжнародною системою класифікації для точних, технічних та природничих наук у світі. Онтології доменів також є прикладами ієрархічних класифікацій, хоча треба зазначити, що онтологію певного домену може бути створено як кілька ієрархій, кожна зі своєю основою;

- пошуковий образ ПВК може складатися з визначених наперед порцій (фасет), кожна з яких належить до окремого аспекту розгляду ПВК. Це є фасетний спосіб. Усередині фасети інформацію можна структурувати на окремі поля, значення яких можуть належати до певної онтології або вільного (наперед не визначеного) списку дескрипторів (термінів), або до одного з наперед визначених альтернативних значень. Тобто, фасетний спосіб може поєднувати в собі можливості двох наведених вище способів.

Кожний із способів, який може бути обрано для побудови інформаційної моделі ПВК, потребує, з одного боку, певного обсягу та ступеня кваліфікації праці, що вкладається попередньо для створення каталогу, а з іншого боку, забезпечує досягнення тієї чи іншої якості пошуку. Будемо вважати критеріями якості пошуку, по-перше, можливість отримати (внаслідок пошуку в каталозі) інформацію, повністю або частково відповідну (кажуть, релевантну) інформаційному запиту, а по-друге, швидкість отримання відповіді.

Спосіб вільного індексування може бути реалізовано з мінімальними витратами праці фахівців при мінімальній кваліфікації. ПВК, що вносяться до каталогу, можуть індексуватися автоматично тими термінами, які згадуються в їхніх описах. Але при цьому є імовірність, що терміни, які було використано для опису ПВК її автором, і терміни, котрі потенційний користувач буде вживати для вираження своєї потреби під час пошуку підходящої ПВК, не збігаються через розбіжність та неузгодженість онтологій, якими користується кожна з діючих сторін.

Спосіб ієрархічних класифікацій потребує попереднього проведення аналізу проблемних галузей експертами високої кваліфікації, здатними побудувати онтології, що відображають специфіку завдань доменів. Отримані ними онтології можуть служити базою для утворення ієрархічних меню, котрі керують формулюванням запитів на пошук ПВК. У цьому разі істотно знижується ефект неузгоджених термінів, а пошук зводиться до вибору однієї з поданих можливостей. Однак цей підхід потребує інвестицій як для початкової побудови онтологій, так і для супроводження та модифікації їх.

Фасетний спосіб також вимагає попередньої експертної підготовки. До його переваг слід віднести можливість звууження кола пошуку за рахунок багатоаспектних критеріїв пошуку.

Слід зазначити, що витрати на побудову онтологій має бути виправдано успішним пошуком, без якого взагалі немає сенсу говорити про повторне використання. Справді, нині маємо бібліотеки модулів мовами програмування, які налічують тисячі об'єктів зберігання. При цьому їх каталоги структуровані переважно лише назвами доменів використання (наприклад, обчислювальні функції, функції роботи з екранами, функції введення-виведення тощо), до того ж каталоги бібліотек фірм-розробників можуть використовувати різні назви для однакових доменів.

На сьогодні існує досить значна множина доменів, досвід роботи з якими дозволяє визначити їхні онтології, стандартизація котрих потребує переважно організаційних зусиль, після чого поняття, що входять до їхнього складу, можуть однозначно трактуватися. Саме така робота тепер проводиться у світі [11].

Джерелом знань для побудови онтологій можуть бути формалізовані вимоги на розробку ПВК як програмних систем, способи отримання яких розглянуто нами в главах 3,4,5.

Наявність спектра моделей для відображення різних аспектів вимог (властивості класів об'єктів, їхніх зв'язків, динаміка поведінки тощо) дає підстави для поділу пошукового образу ПВК на фасети. Ще одним джерелом виявлення фасет є традиції побудови окремих інструментів для окремих аспектів створення програмних систем (таких, як інтерфейси користувача, композиції компонент, розподілена обробка тощо), що дозволяє ставити у відповідність таким аспектам окремі фасети пошукового образу ПВК. Такий підхід дає можливість визначити незалежні напрями адаптації ПВК і спростити розуміння їхніх властивостей.

Інформаційно-пошукові системи зберігання й пошуку пошукових образів ПВК мають спеціальну назву *репозиторіїв* компонент повторного використання.

## Контрольні запитання і завдання

1. Назвіть дві характерні властивості повторно використовуваних компонент (ПВК).
2. Назвіть головні процеси життєвого циклу розробки, властиві систематичному повторному використанню.
3. Охарактеризуйте основні властивості ПВК, поєднання яких визначає її категорію.
4. Охарактеризуйте основні властивості ПВК для окремих категорій:
  - а) мов програмування;
  - б) генераторів програм;
  - в) бібліотек програм;
  - г) каркасів;
  - д) абстрактних доменів.
5. Як виявляється й представляється варіантність проектних рішень під час створення ПВК?
6. Якими механізмами конкретизуються варіанти рішень у реальних проектах?
7. Охарактеризуйте підходи до організації пошуку потрібних ПВК.
8. Визначте процес проектування ПВК.
9. Наведіть методи побудови інформаційного забезпечення ПВК.

## Література до глави 12

1. Бабенко Л.П. Повторное использование в программной инженерии // Кибернетика и системный анализ. — 1999. — № 2. — С. 37—48.
2. Sutcliffe A. and Maiden N. The Domain Theory for Requirement Engineering // IEEE Trans. on Software Engrg. — 1998. — V. 24. — № 3. — P. 174—190.
3. Бабенко Л.П. и др. Адаптивные компоненты повторного использования в системах генерации программ // Кибернетика и системный анализ. — 1991. — № 5. — С. 145—159.
4. Сахаров А.А. Принципы проектирования и использования многомерных баз данных // СУБД. — 1996. — № 3. — С. 44—59.
5. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приёмы объектно-ориентированного проектирования. Патерны проектирования — СПб: Питер, 2001. — 368 с.
6. Schmid H.A. Creating Applications from Components: A Manufacturing Framework Design // IEEE Software, 1996.— November. — P. 67—75.
7. Keepence B., Mannioni M. Using patterns in Model Variability in Product Families // IEEE Software — 1999.—№ 3. — P.102—108.
8. Jacobson I., Griss M., Jonsson P. Software Reuse. — N.-Y.: Addison-Wesley, 1997. — 497 p.
9. Бабенко Л.П., Поляничко С.Л. Модели классификации объектов программной инженерии // Проблемы программирования. — Вып. 1. — К.: Ин-т программных систем НАН Украины, 1997. — С. 25—32.
10. Лаврищева Е.М., Грищенко В.Н. Сборочное программирование. — К.: Наук. думка, 1991. — 216 с.
11. <http://www-ksl.stanford.edu.html>
12. Бен-Ари М. Языки программирования. Практический сравнительный анализ. — М.: Мир, 2000. — 366 с.

## Глава 13. СТАНДАРТИ ПРОГРАМНОЇ ІНЖЕНЕРІЇ

Становлення програмної інженерії як масової соціально значущої професії привело до підвищення ролі використання готових компонент для створення програм і проблеми інтеграції їх, що було розглянуто в попередній главі. Якісний поріг такого використання неможливо подолати, якщо не побудувати ефективно діючу систему раціональних стандартів. Системи стандартів ефективно регулюють виробництво більшості галузей народного господарства, фіксуючи показники якості продукції, технологію її виробництва та інші аспекти тиражування продукції.

У програмній інженерії нема проблеми тиражування окремих продуктів, вона розв'язується простим копіюванням. Головне питання в даному разі полягає в тому, щоб запобігти марному витрачання інтелектуальних та матеріальних ресурсів на вибір і стикування способів надання програмних послуг, відмінність яких не має принципового характеру. Інакше кажучи, мова йде про стандартизацію інтерфейсів тих складових, які використовують для побудови програмних систем.

Метою всякої стандартизації є закріплення досягнень технологій, завдяки чому досвід кращих стає практикою середніх, а загальний технологічний рівень підвищується. На шляху до реалізації цієї мети має бути вирішено такі завдання:

- створення понятійного середовища спілкування всіх, хто бере участь у процесах програмної інженерії;
- підвищення фахової дисципліни;
- забезпечення захисту прав покупця, виробника та бізнес-менеджера програмної продукції.

**Аспекти стратегії стандартизації.** Всяка стандартизація базується на компромісі між виробником і споживачем виробів та послуг, причому компроміс має приносити вигоду обом сторонам. Багаторічний досвід стандартизації у сферах людської діяльності дозволяє сформулювати кілька загальних тез, які доцільно використовувати для досягнення такого компромісу.

**Теза 1.** Стандарт має бути прийнятним, але не обов'язково оптимальним. Інакше кажучи, коли є нагальна потреба в стандарті, ліпше мати посередній стандарт, ніж не мати жодного.

**Теза 2.** Стандарт має забезпечувати ефективну взаємодію сторін при мінімальних обмеженнях свободи поведінки кожної з них, тобто сполучати їхню свободу творчості з ефективною взаємодією. Таким чином, стандарт — це переважно модель взаємодії. Взаємодіяти можуть люди, організації, програмні компоненти або системи. Надавати перевагу можна офіційно або неофіційно, добровільно або примусово, інтуїтивно або усвідомлено. Але в усіх наведених ситуаціях має бути стандартизаційна ситуація, яка полягає в усвідомленні таких положень:

- по-перше, є нагальна потреба підвищення ефективності взаємодії;
- по-друге, різноманітність моделей взаємодії нічим не виправдана і є можливість обрати серед них найбільш конкурентоздатну;
- по-третє, є можливість забезпечити для обраної моделі взаємодії економічні та організаційні переваги.

**Теза 3.** Кожний стандарт має свою сферу застосування, в якій він є ефективним. Інколи це предметна галузь (авіоніка, математика, бізнес тощо), в інших випадках — певний комплекс послуг, що належить до визначеного шару архітектури програмних компонент (див. рис. 4.2).

**Теза 4.** Стандарт може принести користь тільки тоді, коли діють чіткі правила перевірки його дотримання, тобто всі вимоги стандарту можна перевірити під час взаємодії сторін, а ще краще — до взаємодії сторін. Як наслідок цієї тези є наступні дві.

**Теза 5.** Не треба ускладнювати стандарт регламентаціями, які неможливо перевірити або порушення яких є не критичним для взаємодії.

**Теза 6.** Умовою введення стандарту має бути наявність апарата контролю за його виконанням.

**Розробники стандартів.** Існує певна кількість організацій, профілюючими напрямками діяльності котрих є розробка й супроводження стандартів. Залежно від сфер поширення (на відміну від згаданої вище сфери застосування) стандарт може мати статус міжнародного, відомчого або стандарту підприємства. Головним органом встановлення міжнародних стандартів є відома Міжнародна організація із стандартизації (The International Standards Organization або скорочено ISO), яка діє у співпраці з Міжнародною електротехнічною комісією (The International Electrotechnical Commission або скорочено IEC). Усі затверджені ними стандарти мають ідентифікатор, що складається з префіксу ISO/IEC, серійного номера стандарту та дати випуску. Наприклад: ISO/IEC 12207:1995-08-11.

Кожний стандарт ISO/IEC має також назву, яка при посиланнях зазначається після ідентифікатора. ISO/IEC має десятки технічних профільних комітетів, зокрема технічний комітет “Інформаційні технології”, одним з технічних підкомітетів якого є підкомітет з програмної інженерії. Діють також міжнародні об'єднання з окремих проблемних галузей, котрі випускають стандарти для відповідних застосувань.

Кожна цивілізована держава має свої національні органи стандартизації. Так, в Україні маємо Державний комітет України з питань стандартизації (Держстандарт України).

Більшість національних комітетів зі стандартизації визнають стандарти ISO/IEC і входять до її складу, але піддають стандарти ISO/IEC так званій процедурі гармонізації, яка полягає в пристосуванні їх до національних умов та особливостей застосування (наприклад, національні алфавіти, метричні системи, валютні знаки тощо).

Потужним джерелом стандартів є професійні об'єднання, зокрема, для програмної інженерії — це IEEE Computer Society.

Є більш як 300 стандартів IEEE з програмної інженерії, значна частина яких береться до уваги широким загалом розробників систем. Практично більшість стандартів програмної інженерії історично з'являються як стандарти IEEE, а згодом, після випробування досвідом використання, вносяться як кандидати в стандарти ISO/IEC.

Треба сказати, що процедура затвердження стандартів ISO досить складна: розрізняють кілька стадій проходження кандидата в стандарт, у кожній з них передбачено розсилання пропозиції на експертизу всім національним комітетам, збирання зауважень, обробка їх та голосування, створення нової версії. Усе це може тривати роками.

Зазначене вище стосується офіційних стандартів, або, як кажуть, стандартів де-юре. На жаль, забюрократизована процедура призводить до того, що стандарти, технічно доведені до кондиції, за час доведення можуть втратити свою значущість для індустрії програмного забезпечення, відповідність діючому рівню технології.

Тим часом індустрія створює так звані стандарти де-факто, які фактично масово використовуються незалежно від того, затверджені вони плановими компетентними органами чи ні, бо відповідають найактуальнішим болючим проблемам індустрії і вирішують такі проблеми “тут і зараз”.

Терміном “стандарт де-факто” часто позначають специфікації, опубліковані деяким консорціумом (групою закладів, котрі офіційно працюють для спільної мети) або фірмою як внутрішній стандарт, але які загалом схвалили інші розробники, і вони стали сприйматися як необхідний сучасний рівень певного напрямку технології. Так, проектні рішення консорціуму “Object Management Group” (OMG) щодо керування транзакціями стали стандартом де-факто. Моделі UML (див. главу 5) фірми “Rational” визначаються як майбутні стандарти моделювання артефактів програмної інженерії. Мова Java активно завойовує ринок.

Стандарти де-факто завдяки своїй відповідності актуальним потребам ринку поширюються і впроваджуються досить швидко. Це можна пояснити тим, що їх опрацювання проходить усередині груп, які їх створюють (іноді на це потрібно багато років), а публікація стосується вже зрілих рішень.

Враховуючи сказане вище, впливові міжнародні організації і в першу чергу ISO, котрі є розробниками стандартів де-юре, визнали, що вони компетентні джерела стандартів, але не монополні, і тому запровадили процес перетворення стандартів де-факто на стандарти де-юре, які в цьому разі назвали загальнодоступною специфікацією (Public-Available Specifications — скорочено PAS).

**Об'єкти стандартизації.** Як було зазначено в главі 2, еталонна модель програмної інженерії визначає такі головні елементи:

- процеси (розробка);
- продукти (розробки);
- ресурси (які використовуються процесом для створення продукту).

Важливим елементом моделювання проблем є також клієнт, який замовляє систему (замовник). Вимоги клієнта визначають склад та якість вказаних попередньо елементів. Тож об'єктами стандартизації є аспекти вказаних вище елементів або поєднання їх.

Найбільше стандартів є для процесів. Треба зазначити при цьому, що вдосконалення процесів, як правило, веде до вдосконалення продуктів та ефективного використання ресурсів.

Визначено детальний перелік процесів та дій, що складають процеси, для всіх етапів життєвого циклу розробки та більшості аспектів названих етапів. Найбільшого успіху домогся і став широко використовуватися стандарт [2] для процесів життєвого циклу програмного забезпечення, який розглянуто у главі 2. Цей стандарт встановив певний каркас для розгляду всіх проблем програмної інженерії (інколи його називають парасолькою, що накрила всі питання щодо замовника та процесів реалізації його замовлення).

Інші стандарти для процесів визначаються переважно в термінах дій, визначених “парасолькою”.

Першим виміром класифікації є відношення стандарту до продукту, процесу, ресурсу чи взаємодій з клієнтом.

У джерелі [3] запропоновано другий вимір класифікації стандартів за рівнями узагальнення регламентацій, які подаються в стандарті.

Окремі групи становлять:

- визначення термінології;
- загальні рекомендації;
- принципи дій;
- стандартизовані елементи;
- рекомендації для прикладних застосувань;
- рекомендації щодо інструментів та методів розробки (наприклад, класифікація аномалій виконання);
- плани керування якістю;
- плани валідації та верифікації.

Згідно із запропонованою класифікацією в джерелі [3] наведено найбільш значущі діючі стандарти з програмної інженерії.

## Контрольні запитання і завдання

1. Визначте основні тези стандартизації в програмній інженерії.
2. Які є міжнародні організації зі створення стандартів в галузі програмного забезпечення?

## Література до глави 13

1. Александров А.Л., Бабенко Л.П., Кауфман В.П., Ющенко Е.Л. Стандартизация языков программирования. — К.: Техніка, 1989. — 160 с.
2. ISO/IEC 12207:1995-08-01: Informational Technology — Software life cycle processes.
3. Moore J.W. An Integrated Collection of Software Engineering Standards, IEEE Software. — 1999. — November/December. — P. 51—57.

## ДОДАТОК А

### Етичний кодекс програмної інженерії

Є потужні професійні об'єднання фахівців з програмної інженерії. Серед них у першу чергу треба назвати The Association for Computer Machinery (ACM) та Institute of Electrical and Electronics Engineers Computer Society (IEEE Computer Society). Спільними зусиллями цих двох об'єднань розроблено кодекс етики програмної інженерії, який фіксує мораль, правила та норми поведінки відповідних професіоналів, їхні зобов'язання й відповідальність перед суспільством та один перед одним [1].

Треба сказати, що етика інженерної діяльності дещо відмінна від етики прикладних досліджень. Дослідники, котрі працюють у прикладній науці, спрямовують свої зусилля для реалізації можливостей, що відповідають певним вимогам.

Інженерна діяльність, окрім зазначеного вище, має певну традицію, котра полягає в поєднанні технічних умінь з відповідальністю перед користувачами та громадськістю, з умінням керування, що приводить до вдалого завершення великих проектів, за якими створюються безпечні ефективні системи. Інакше кажучи, інженери мають добре знати різницю між ризиками, яка є між швидкоствореним прототипом і високоякісною реалізацією з низьким ризиком.

Зазначимо, що швидке внесення змін потребує додаткових умов щодо сертифікації отриманих результатів.

Кодекс складається з преамбули і восьми принципів, дотримання яких обов'язкове для професіоналів з програмної інженерії. У преамбулі професіонал визначається як такий, що бере безпосередню участь у діяльності або навчанні з аналізу, специфікації, проектування, розробки, сертифікації, супроводження й тестування програмних систем.

Сформульовані принципи декларують здоров'я, безпеку та добробут суспільства як головні чинники, котрі необхідно брати до уваги при прийнятті рішень у професійній діяльності, яку ми розглядаємо.

Тож задекларовані кодексом вісім принципів відповідно стосуються:

- 1) узгодження професійної діяльності з інтересами суспільства;

- 2) взаємовідносин між клієнтом, роботодавцем і виконавцем розробки;
- 3) досягнення відповідності якості продукту кращим професійним стандартам;
- 4) дотримання чесності й незалежності при професійних оцінках;
- 5) дотримання етичних норм у менеджменті і супроводженні розробок;
- 6) підтримання становлення професії відповідно до кодексу етики;
- 7) дотримання етичних норм у взаємовідносинах між колегами;
- 8) самовдосконалення з фаху.

Кожний з наведених принципів докладно пояснюється щодо багатьох аспектів його дотримання.

## Література

1. Jotterbarn D., Miller K., Rogerson S. Software Engineering Code of Ethics is Approved // Communications of the ACM. — 1999. — V. 42. — № 10. — P. 102—107.
2. [www. asm. org / serving / se / code. hfm](http://www.asm.org/serving/se/code.hfm)

---

## ДОДАТОК Б

### Глосарій

**Абстрактна архітектура** — декомпозиція рішення для виділеного спектра завдань домену на підсистеми або ієрархію підсистем, на кожному рівні якої фіксуються можливі варіанти виділених параметрів і обмежень, що визначають відповідні варіації складу виділених компонент.

**Агрегація** — об'єднання кількох понять у нове поняття, істотні ознаки нового поняття при цьому можуть бути або сумою ознак компонент або суттєво новими (відношення агрегації часто ототожнюють з відношенням “частка — ціле”).

**Адаптивність** — атрибут, що характеризує зусилля, необхідні для настроювання програм до різних середовищ без виконання особливих дій або без засобів.

**Активні домени** — домени, які можуть переходити від стану до стану без зовнішніх стимулів, як це робиться з людиною чи атмосферою.

**Актори** — чинні особи, для яких створюється система.

**Аналіз вимог** — відображення функцій системи та її обмежень у моделі проблеми.

**Аналізованість** — показник супроводжуваності, який визначає необхідні зусилля для діагностики випадків відмов або ідентифікації частин програм, що будуть модифікуватися.

**Артефакт** — будь-який продукт діяльності фахівців з розробки програмного забезпечення.

**Архітектура програмної системи** — визначення системи в термінах обчислюваних складових (підсистем) та інтерфейсів між ними, яке відображає правила декомпозиції проблеми на складові.

**Асоціація** — найзагальніше відношення, що утворює наявність зв'язку між поняттями, не уточнюючи залежності їхнього змісту та обсягів.

**Асоціація альтернативна** — можливість для деякого класу перебувати одночасно у зв'язку з одним елементом з певної множини класів.

**Безвідмовність** — атрибут надійності, що визначає відсутність або частоту відмов через помилки в програмному забезпеченні.

**“Білої скриньки” метод** — досліджування внутрішньої структури програми для виявлення всіх помилок відповідно до критерію вичерпного тестування маршрутів та потоків передач керування.

**Валідація** — забезпечення відповідності розробки вимогам її замовників.

**Верифікація** — перевірка правильності трансформації проекту в код реалізації.

**Взаємодія об’єктів** — обмін повідомленнями між об’єктами.

**Використання програмного забезпечення** — діалогове спілкування користувача із системою або недіалогове функціонування для одержання результатів у різноманітних середовищах.

**Вимога** — угода або договір між замовником і виконавцем системи, які стосуються її роботи.

**Відмова (failure)** — відхилення програми від функціонування або неможливість виконувати функції, визначені вимогами та обмеженнями, тобто перехід програми в непрацездатний стан у зв’язку з помилками у програмі або аварією в середовищі функціонування.

**Відновлюваність** — атрибут надійності, який вказує на спроможність програми до перезапуску для повторного виконання і відновлення даних після відмов.

**Відношення використання сценарію** — включення до сценарію іншого сценарію повністю (без настроювання).

**Відношення розширення сценарію** — деталізація й нарощування новими функціями визначеного раніше сценарію.

**Водоспадна або каскадна модель** — схема робіт, за якою кожна з робіт виконується один раз і в тому порядку, в якому роботи перелічено в моделі життєвого циклу.

**Гарантія (підтвердження) якості програмного забезпечення** — дії на кожній стадії життєвого циклу з перевірки і підтвердження відповідності стандартам та процедурам досягнення якості.

**Дефект (fault)** — результат помилок розробника у вхідних або проектних специфікаціях, текстах кодів програм, експлуатаційній документації тощо.

**Динамічні методи тестування** — виконання програм з метою встановлення причин помилок з очікуваними реакціями на ці помилки.

**Діаграма** — графічне представлення елементів моделювання системи (класів, сценаріїв, поведінки, послідовності, співробітництва, активності, станів, реалізації, компонент тощо).

**Домен предметної галузі** — спектр завдань предметної галузі, які допускають схожі способи їх вирішення.

**Екземпляризація** — залежність між параметризованим абстрактним класом-шаблоном (template) і реальним класом, який ініційовано шляхом визначення параметрів шаблону.

**Експлуатація або супроводження** — дії з використання готової програмної системи.

**Ефективність ресурсів** — атрибути, які показують кількість ресурсів і тривалість використання їх при виконанні функцій програми в програмному забезпеченні.

**Ефективність системи** — множина атрибутів раціональності, які показують взаємозв’язок між рівнем виконання програмного забезпечення і кількістю використовуваних ресурсів (засоби, апаратура та ін.).

**Забезпечення гарантії якості** — дії з перевірки й підтвердження відповідності програмного забезпечення стандартам і процедурам, орієнтованим на досягнення якості.

**Загроза** — прояв нестійкості, що порушує безпеку системи.

**Залежність класів** — використання певного сервісу іншого класу.

**Захищеність** — атрибут функціональності, який вказує на можливість запобігати несанкціонованому доступу (випадковому або навмисному) до програм і даних.

**Змінюваність** — показник супроводжуваності, який визначає зусилля, які витрачаються на модифікування, видалення помилок або внесення змін у зв’язку з помилками або новими умовами середовища функціонування.

**Зрозумілість** — атрибут зручності, який визначає зусилля, необхідні для розпізнавання логічних концепцій та умов їх застосування.

**Зручність застосування** — множина властивостей ергономічності, котрі полегшують освоєння й адаптацію програм до умов експлуатації, що змінюються й зумовлюють легкість підготовки даних, зрозумілість результатів, зручність внесення змін і вказують на сприятливі умови для використання програмного забезпечення певним колом користувачів.

**Інертні домени** — домени, зміна станів яких ніколи не відбувається з ініціативи об’єктів домену, а реалізується під дією зовнішніх факторів.

**Інженерія** — застосування наукових результатів, яке дозволяє отримувати користь від властивостей матеріалів та джерел енергії.

**Інженерія вимог** — збір, аналіз вимог замовника та представлення їх у нотації, яка є зрозумілою й узгодженою як замовником, так і виконавцем.

**Інженерія якості** — процес надання продуктам програмного забезпечення властивостей якості, оцінки показників та кількісного вимірювання їх з зіставленням із зазначеними вимогами.

**Інспекція коду** — формальна перевірка відповідності опису використовуваних типів і структур даних проекту системи поставленим до системи вимогам.

**Інтервал стабільності** — інтервал часу, протягом якого не відбувається зміна стану.

**Інтероперабельність** — атрибут функціональності, який забезпечує спроможність взаємозамінювання спільно діючих компонентів у

системи програм з необхідною інсталяцією або адаптацією всього програмного забезпечення.

**Інформаційна модель ПВК** — модель, за допомогою якої відображається інформаційний образ окремої компоненти в каталозі (так званий пошуковий образ).

**Інформаційна система** — система, яка проводить збір, обробку, зберігання та вироблення інформації людьми з використанням автоматичних процесорів.

**Інформаційне забезпечення повторного використання** — апарат інформування потенційних виконавців нових розробок про існування, зміст та умови застосування наявних ПВК.

**Інцидент** — абстрактна подія, яка впливає на зміну стану об'єкта.

**Кадастр** — інтегровані засоби накопичення повторно використовуваних компонент (ПВК) і пошуку їх для подальшого використання.

**Каркаси (фрейми)** — різновид абстрактних архітектур або частково визначена абстрактна архітектура з відкладеним до визначенням виділених компонент.

**Керування якістю** — комплекс засобів та системної діяльності з планування, забезпечення й оцінювання якості програмного забезпечення.

**Кінцеві користувачі** — особи, для фахових потреб яких замовляється комп'ютерна система.

**Когнітивна відстань** — кількість інтелектуальних зусиль, які необхідно затратити розробникові програмного забезпечення для того, щоб перевести систему, що розробляється, з однієї стадії життєвого циклу в іншу.

**Кодування** — див.: **Реалізація програмної системи.**

**Компонента** — тип, клас, проектне рішення, документація або інший продукт програмної інженерії, спеціально пристосований для повторного використання.

**Компонентна розробка** — конструювання програмного забезпечення з конструкторів за каталогом як композиції готових компонентів.

**Конкретизація** — додавання істотних ознак, завдяки чому зміст поняття розширюється, а обсяг поняття звужується.

**Конструювання** — див.: **Реалізація програмної системи.**

**Конфігурація** — склад програмної системи, створений шляхом вибору окремих примірників із визначеного заздалегідь набору варіантів модулів.

**Концептуальне моделювання** — процес побудови моделі проблеми, орієнтованої на її розуміння людиною.

**Легкість навчання** — атрибути зручності, котрі вимірюються зусиллями, затраченими користувачами для вивчення умов використання, до яких належать операційний контроль, впровадження, діагностика, а також процедури, документація і правила застосування програмного забезпечення.

**Метрика програм** — система визначених кількісних показників програм та шкали вимірювання їх.

**Модель життєвого циклу** — типова схема послідовності робіт на етапах розробки програмного продукту.

**Модель процесів** — визначення певних дій, які супроводжують зміни станів об'єктів.

**Модель станів** — відображення динаміки змін стану кожного з класу об'єктів, які змінюють свою поведінку.

**Модель функції-дані** — декомпозиція проблеми на послідовність функцій та даних, які обробляються цими функціями.

**Модель якості** — чотирирівнева модель, яка відображає характеристики, атрибути (показники), метрики, оціночні елементи програмного забезпечення.

**Надійність** — множина кількісних атрибутів, яка вказує на спроможність програмного забезпечення перетворювати вхідні дані у вихідні результати за умов, що залежать від періоду часу (зношення й старіння життя програмного забезпечення не враховується).

**Налагодження** — перевірка програмного об'єкта на наявність у ньому помилок і наступне усунення їх без внесення нових помилок.

**Нефункціональні вимоги** — вимоги, які характеризують організаційні, виконавчі, операційні аспекти роботи програмної системи або аспекти середовища реалізації.

**Об'єкти інтерфейсу в методі Джекобсона** — об'єкти сценарію, функцією яких є трансформація повідомлень зовнішніх об'єктів у внутрішні стани сценарію.

**Об'єкти керування в методі Джекобсона** — об'єкти, які здійснюють функції перетворення об'єктів інтерфейсу в об'єкти сутності; часто відображають алгоритми обробки даних у системі.

**Об'єкти-сутності в методі Джекобсона** — довгоживучі об'єкти, які відповідають сутностям реального світу і зберігають свій стан після виконання сценарію.

**Об'єктно-орієнтована модель** — представлення програм системи як сукупності об'єктів, які взаємодіють між собою і мають певні властивості та поведінку.

**Онтологія** — сукупність примітивних понять, термінології і парадигми інтерпретації у сфері проблем, що мають бути розв'язані новою розробкою.

**Оперативність** — атрибути зручності, які характеризують рівень реакції системи на зусилля користувача при виконанні операцій та її операційного контролю.

**Оціночний елемент метрики** — кількісна або якісна оцінка відповідного показника з урахуванням його ваги в системі оцінки якості.

**Оцінювання якості** — дії, спрямовані на визначення ступеня задоволення програмного забезпечення потребами відповідно до їхнього призначення.

**Пакет (у UML)** — загальний механізм організації елементів (об'єктів, класів) у групи, починаючи від усієї системи в цілому (стеріотип “система”) і до підсистем різних рівнів деталізації.

**Переносність** — група властивостей, яка забезпечує пристосовність до переносу з одного середовища функціонування в інші, а також зусилля для перенесення програмного забезпечення до нового середовища або мережі.

**План тестування** — опис стратегії, ресурсів і графіка тестування окремих компонент та системи в цілому.

**Поведінка домену** — зміна станів з плином часу.

**Повна функціональність системи (у UML)** — всі сценарії всіх акторів, виділених у системі.

**Повторно використовувана компонента (ПВК)** — фрагмент знань про минулий досвід розробки систем програмування, який можна адаптувати для створення нових систем і поданий так, що його можуть використовувати не лише його розробники.

**Повторне використання** — використання для нових розробок будь-яких порцій формалізованих знань, отриманих під час реалізації завершених розробок програмних систем.

**Подія** — явище, що провокує зміну певного стану та перехід до іншого стану в домені або системі.

**Помилка** — недоліки в операторах програми або в технологічному процесі її розробки, що призводять до неправильної інтерпретації вхідної інформації, а отже, і до неправильного рішення.

**Правильність (точність)** — атрибут функціональності, що показує, як забезпечується досягнення теоретично правильних та узгоджених результатів.

**Прикладна система** — кінцевий продукт програмної інженерії, призначений для виконання конкретних сценаріїв кінцевого користувача.

**Програмна інженерія** — система методів, здатна до масового відтворення засобів та дисципліни планування, розробки, експлуатації та супроводження програмного забезпечення.

**Програмування** — див: **Реалізація програмної системи**.

**Процес експлуатації** — дії з обслуговування системи під час її використання — консультації користувачів, вивчення їхніх побажань тощо.

**Процес постачання** — дії під час передачі розробленого продукту покупцю.

**Процес придбання** — дії, що ініціюють життєвий цикл системи і визначають організацію-покупця автоматизованої системи, програмної системи чи сервісу.

**Процес розробки** — дії організації-розробника програмного продукту, які включають інженерію вимог до системи, проектування, кодування та тестування.

**Процес супроводження** — дії щодо програмних систем з керування модифікаціями, підтримки актуального стану та функціональної придатності, інсталяції або вилучення версій у користувача.

**Проектування** — перетворення вимог у послідовність проектних рішень щодо системи.

**Проектування архітектурне** — визначення головних структурних особливостей системи, яку будують.

**Проектування детальне** — визначення подробиць функціонування та зв'язків для усіх компонент системи.

**Проектування концептуальне** — уточнення розуміння й узгодження деталей вимог.

**Проектування технічне** — відображення вимог середовища функціонування та розробки системи, визначення усіх конструкцій як композицій компонент.

**Рациональність** — група властивостей, яка характеризує ступінь відповідності використовуваних ресурсів середовища функціонування рівню надійності виконання при заданих умовах застосування програмного забезпечення.

**Реактивні домени** — домени, зміна стану яких відбувається у зв'язку з відповіддю на певну зовнішню подію.

**Реалізація програмної системи** — перетворення проектних рішень у працездатну програмну систему, що реалізує означені рішення (синоніми: кодування, конструювання, програмування).

**Ризик** — сукупність загроз, які можуть призвести до несприятливих наслідків і збитку, а також до змін стану системи або середовища.

**Рівень якості** — відносна характеристика якості, яка базується на порівнянні фактичних значень показників якості оцінюваного програмного забезпечення з базовими значеннями кращих аналогів або зі встановленими вимогами на розробку програмного забезпечення.

**Родові знання про проблему** — знання про певне сімейство задач, які подано у вигляді, придатному для використання при розв'язанні будь-якої задачі, що належить до даного сімейства.

**Сертифікація програмного продукту** — процес, який засвідчує відповідність ідентифікованої програмної продукції (процесу або послуг) конкретному стандарту або технічним умовам з оцінкою спеціальним знаком або свідомством.

**Система компонент** — сукупність взаємоузгоджених компонент, придатна для повторного використання.

**Систематичне повторне використання** — систематична і цілеспрямована діяльність із створення і використання ПВК.

**Система якості** — набір організаційних структур, методик, процесів та ресурсів для здійснення керуванням якістю.

**Сімейства прикладних систем** — множина прикладних систем зі спільними функціональними властивостями.

**Специфікація** — представлення правил, стандартів, критеріїв якості, обмежень на користування об'єктів.

**Спіральна модель** — модель процесів життєвого циклу розробки, яка дозволяє повертатися до будь-якого попереднього процесу життєвого циклу з метою переробки елементів зробленого продукту.

**Стабільність** — атрибути супроводжуваності, які вказують на ризик модифікації системи.

**Стан (домену, системи, об'єкта тощо)** — фіксація певних властивостей на певний момент або інтервал часу.

**Статичні методи тестування** — аналіз та розгляд специфікацій компонентів без виконання їх.

**Стереотип** — показник категорії елемента моделювання UML.

**Стійкість** — атрибут надійності, який вказує на спроможність компоненти виконувати функції в аномальних умовах (збої апаратури, помилки в даних та інтерфейсах, порушення в діях оператора тощо).

**Сумісність** — показник переносності, який визначає можливість використання нових програм у середовищі діючого програмного забезпечення.

**Супроводження** — будь-які роботи із внесення змін до програмної системи після того, як її було передано користувачеві для експлуатації.

**Супроводжуваність** — група властивостей, які вказують на необхідні зусилля для проведення модифікацій, внесення додаткових функцій з відповідним коригуванням програмного забезпечення при зміні середовища, вимог або функціональних специфікацій.

**Сценарій** — один з можливих шляхів використання системи.

**Тест** — програма, призначена для перевірки працездатності іншої програми і виявлення в ній помилкових ситуацій.

**Тестові дані** — дані, які готуються певними генераторами на основі документів програм або користувачем за специфікаціями вимог і які використовуються для перевірки роботи програмної системи.

**Тестовність** — атрибут програмного забезпечення, який вказує на ступінь проведення валідації та верифікації для виявлення помилок у програмах, невідповідностей їх вимогам та виконанням модифікаціям.

**Тестування** — спосіб семантичного налагодження (перевірки) програми, що полягає в опрацюванні програмою послідовності контрольних наборів тестів, для котрих відомий результат.

**Узагальнення** — звуження істотних ознак поняття, що розширює коло охоплених поняттям об'єктів, тобто його обсяг.

**Узгоджуваність** — показник якості програмного забезпечення, який вказує на відповідність заданим стандартам, угодам, правилам, законам і розпорядженням.

**Успадкована система** — діюча система, створена за небажаними або навіть невідомими для команди розробників методами і техноло-

гіями проектування, котру використовують для керування та підтримки процесів бізнесу.

**Успадкування** — конкретизація в підкласі окремих властивостей та поведінки, визначених у суперкласі.

**Фасад (в UML)** — механізм доступу до тих властивостей системи компонент, які потрібні для ревикористання, або погляд на системі компонент того, хто її використовує.

**Функціональні вимоги** — вимоги, які визначають складові мети і функції системи.

**Функціональна повнота** — атрибут, який показує ступінь достатності основних функцій для розв'язання спеціальних задач відповідно до призначення програмного забезпечення.

**Функціональність** — це сукупність властивостей, які визначають спроможність програмного забезпечення виконувати перелік функцій у заданому середовищі, котрі відповідають вимогам.

**Характеристики якості** — функціональність (functionality), надійність (reliability), зручність (usability), ефективність (efficiency), супроводжуваність (maintainability), переносність (portability) тощо.

**Цілісність** — спроможність системи зберігати стабільність у роботі і не мати ризику.

**“Чорної скриньки” метод** — тестування реалізованих функцій шляхом перевірки невідповідності між реальною поведінкою функцій й очікуваною поведінкою згідно з вхідними специфікаціями вимог.

**UML (Unified Modeling Language)** — мова для специфікації, візуалізації, конструювання та документування артефактів програмних систем, а також для моделювання бізнесу.

**Якість програмного забезпечення** — сукупність властивостей, які визначають спроможність програмного забезпечення задовольнити замовника, котрий сформулював ці властивості як вимоги до розробки.

Навчальне видання

Серія “Вища освіта ХХІ століття”

*БАБЕНКО Людмила Петрівна*  
*ЛАВРІЩЕВА Катерина Михайлівна*

## ОСНОВИ ПРОГРАМНОЇ ІНЖЕНЕРІЇ

Навчальний посібник

Керівник видавничих проектів *В.І. Карасьов*  
Головний редактор *В.П. Розумний*  
Контрольний редактор *О.М. Гречишкіна*  
Коректор *Ю.А. Полулященко*  
Комп'ютерна верстка *Д.А. Грека*  
Обкладинка *В.С. Жиборовського*

Підп. до друку 25.06.01. Формат 60×84 <sup>1</sup>/<sub>16</sub>.  
Папір офс. Друк офс. Гарнітура шкільна.  
Ум. друк. арк. 15,8. Обл.-вид. арк. 18,9.  
Наклад 1000 прим. Зам. № 1-2294

Київська обласна організація товариства “Знання” України  
01034, м. Київ-34, вул. Стрілецька, 28  
Свідоцтво ДК №125 від 20.07.2000  
Тел. (044) 224-80-43, 224-23-36  
E-mail: sales@society.kiev.ua  
<http://www.znannia.com.ua>

ЗАТ “Віпол”. ДК № 15  
03151, м. Київ, вул. Волинська, 60

## Книгу можна придбати за адресами:

- ✓ м. Київ, вул. Володимирська, 42, “Дніпро”, тел. (044)224-92-40;
- ✓ м. Київ, вул. М. Грушевського, 4, “Наукова думка”,  
тел. (044)228-06-96;
- ✓ м. Київ, пл. Лесі Українки, 1, тел. (044)296-80-20;
- ✓ м. Вінниця, вул. Соборна, 38, “Дружба”, тел. (0432)32-76-93;
- ✓ м. Дніпропетровськ, вул. Дзержинського, 1/3, СП “Мир”,  
тел. (0562)46-52-32;
- ✓ м. Донецьк, вул. Артема, 147-а, “Дом книги”, тел. (0622)55-74-49;
- ✓ м. Житомир, вул. Київська, 59, “Знання”, тел. (0412)37-27-42;
- ✓ м. Івано-Франківськ, вул. Вічевий майдан, 3, “Сучасна українська книга”,  
тел. (03422)3-04-60;
- ✓ м. Івано-Франківськ, вул. Незалежності, 19, “Букініст”,  
тел. (03422)2-38-28;
- ✓ м. Запоріжжя, просп. Леніна, 151, “Сучасник”, тел. (0612)33-12-27;
- ✓ м. Кіровоград, вул. К. Маркса, 34/25, “Книги”, тел. (0522)22-48-45;
- ✓ м. Кривий Ріг, пл. Визволення, 1, “Букініст-салон”,  
тел. (0564)29-81-21;
- ✓ м. Львів, просп. Шевченка, 8, книгарня НТШ, тел. (0322)79-85-80;
- ✓ м. Львів, просп. Шевченка, 16, ТОВ “Ноти”, тел. (0322)72-67-96;
- ✓ м. Луганськ, вул. Радянська, 58, “Глобус-книга”,  
тел. (0642)53-62-30;
- ✓ м. Луцьк, просп. Волі, 8, “Освіта”, тел. (03322)7-46-14;
- ✓ м. Одеса, вул. Троїцька, 28, “Передплатні видання”, ООО “Епос”,  
тел. (0482)22-10-80;
- ✓ м. Полтава, вул. Жовтнева, 60-а, “Планета”, тел. (05322)7-20-19;
- ✓ м. Рівне, вул. Соборна, 57, “Слово”, тел. (0362)26-94-17;
- ✓ м. Сімферополь, вул. Куйбишева, 13, МСП “Ната”,  
тел. (0652)25-62-16;
- ✓ м. Суми, вул. Кірова, 8, АТП “Кобзар”, тел. (0542)22-51-17;
- ✓ м. Тернопіль, вул. Танцорова, 11, “Навчальна книга — Богдан”,  
тел. (0352)25-18-09, 43-00-46, 25-37-53, 25-28-41;
- ✓ м. Харків, вул. Петровського, 6/8, “Вища школа”,  
тел. (0572)47-80-20;
- ✓ м. Херсон, вул. 9-го січня, 15, офіс 75, ПВКФ “Стиль-Плюс”,  
тел. (0552)24-22-96;
- ✓ м. Хмельницький, вул. Подільська, 25, “Книжковий світ”,  
тел. (03822)6-60-73;
- ✓ м. Чернівці, вул. О. Кобилянської, 37, “Художня книга”,  
тел. (03722)2-60-95;
- ✓ м. Чернігів, вул. Леніна, 45, “Будинок книги”, тел. (046)227-30-03.

Книготорговельним організаціям та оптовим покупцям звертатися  
за тел.: (044) 238-82-62, 224-80-43; факс: 238-82-68.  
E-mail: sales@society.kiev.ua <http://www.books.com.ua>

# ВИДАВНИЦТВО “ЗНАННЯ” ПРОПОНУЄ

НБ ПНУС



637782

**Барроу К., Барроу П., Браун Р. Бізнес-план: Практик. посіб.** — Пер. з 3-го англ. вид. — К.: Т-во “Знання”, КОО, 2001. — 285 с. — Мова укр. — Формат 60×84 <sup>1</sup>/<sub>16</sub>. — Обкл. м’яка.  
ISBN 966-620-051-1 (Укр.)  
ISBN 0-7494-2696-9 (Англ.)

Це перше видання українською мовою добре відомого фахівцем практичного посібника із бізнес-планування. Він витримав видання чотирма мовами й успішно використовується тисячами читачів, які зацікавлені в тому, щоб навчитися складати бізнес-плани для підприємств, що тільки створюються або розвиваються. Посібник створено на основі Програми з підприємництва Кренфілдської школи менеджменту. У ньому в доступній формі даються практичні професійні поради, як приступити до пошуків та підготовки матеріалів для складання бізнес-плану, розкривається, яким чином у бізнес-плані мають бути відображені маркетинг, фінанси та законодавство, наводиться 20 завдань з діаграмами, схемами та живими прикладами з реальних бізнес-планів. Переклад здійснено з третього, переробленого і доповненого англійського видання.

Для керівників підприємств і організацій, підприємців, а також для студентів економічних спеціальностей, викладачів, слухачів шкіл бізнесу.

**Книготорговельним організаціям та оптовим покупцям  
звертатися за тел.: (044) 238-82-62, 224-80-43.  
E-mail: [sales@society.kiev.ua](mailto:sales@society.kiev.ua) <http://www.books.com.ua>**

**[www.BOOKS.com.ua](http://www.BOOKS.com.ua)**

Широкий вибір навчальної та ділової літератури

Тел. для довідок: (044) 235-00-44, 224-80-43